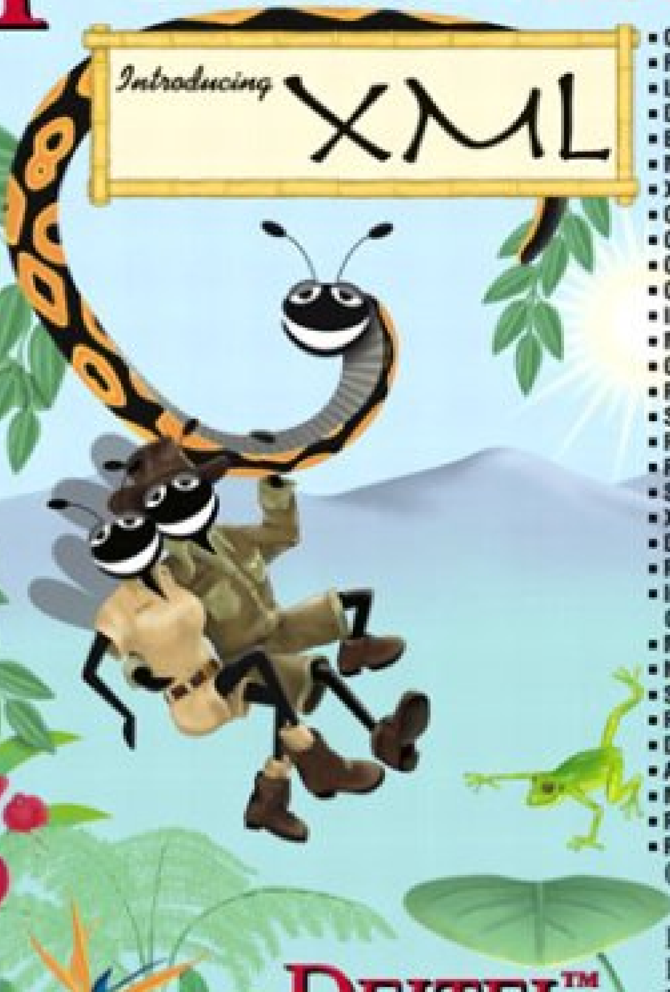
PYTHON HOW TO PROGRAM

Introducing XML



- Control Structures
- Functions
- Lists and Tuples
- Dictionaries
- Exceptions
- Modules
- XHTML™/CSS™
- CGI
- Classes
- Class Attributes
- Class Customization
- Inheritance
- Method Overriding
- GUI/Tkinter
- Python Mega Widgets
- String Manipulation
- Regular Expressions
- File Processing
- Serialization
- XML Processing
- Databases/DB-API/SQL
- Process Management
- Interprocess Communication
- Multithreading
- Networking/Sockets
- Security
- Restricted Execution
- Data Structures
- Accessibility
- Multimedia
- PyOpenGL
- Python Server Pages (PSP)

DEITEL
DEITEL
LIPERI
WIEDERMANN

DEITEL™

Python How to Program, 1/e
Table of Contents

1

Introduction to Computers, Internet and World Wide Web

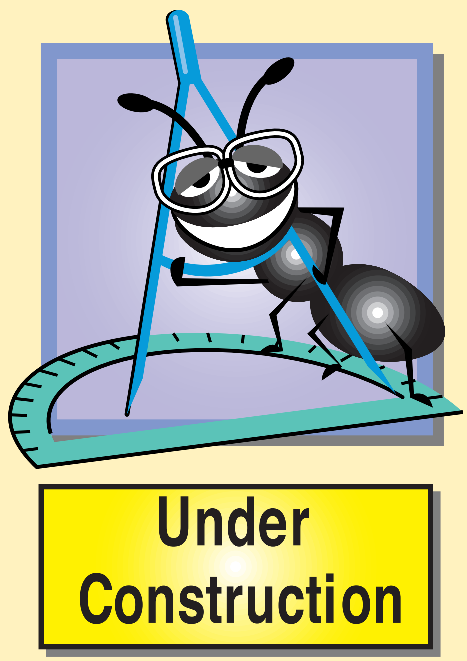
Objectives

- To understand basic computer concepts.
- To become familiar with different types of programming languages.
- To become familiar with the history of the Python programming language.
- To preview the remaining chapters of the book.

Things are always at their best in their beginning.
Blaise Pascal

High thoughts must have high language.
Aristophanes

Our life is frittered away by detail…Simplify, simplify.
Henry David Thoreau



Outline

1.1 Introduction

Welcome to Python! We have worked hard to create what we hope will be an informative and entertaining learning experience for you. The manner in which we approached this topic created a book that is unique among Python textbooks for many reasons. For instance, we introduce early in the text the use of Python with the *Common Gateway Interface (CGI)* for programming Web-based applications. We do this so that we can demonstrate a variety of dynamic, Web-based applications in the remainder of the book. This text also introduces a range of topics, including *object-oriented programming (OOP)*, the Python *database application programming interface (DB-API)*, graphics, the *Extensible Markup Language (XML)*, security and an appendix on Web accessibility that addresses programming and technologies relevant to people with impairments. Whether you are a novice or an experienced programmer, there is much here to inform, entertain and challenge you.

Python How to Program is designed to be appropriate for readers at all levels, from practicing programmers to individuals with little or no programming experience. How can one book appeal to both novices and skilled programmers? The core of this book emphasizes achieving program clarity through proven techniques of *structured programming* and

object-based programming. Nonprogrammers learn basic skills that underlie good programming; experienced programmers receive a rigorous explanation of the language and may improve their programming styles. To aid beginning programmers, we have written this text in a clear and straightforward manner, with abundant illustrations. Perhaps most importantly, the book presents hundreds of complete working Python programs and shows the outputs produced when those programs are run on a computer. We call this our *Live-Code™ approach.* All of the book's examples are available on the CD-ROM that accompanies this book and on our Web site, **`www.deitel.com`**.

Most people are at least somewhat familiar with the exciting capabilities of computers. Using this textbook, you will learn how to command computers to exercise those capabilities. It is *software* (i.e., the instructions you write to command the computer to perform *actions* and make *decisions*) that controls computers (often referred to as *hardware*).

Computer use is increasing in almost every field. In an era of steadily rising costs, the expense of owning a computer has been decreasing dramatically due to rapid developments in both hardware and software technology. Computers that filled large rooms and cost millions of dollars 25 to 30 years ago now are inscribed on the surfaces of silicon chips smaller than a fingernail and that cost perhaps a few dollars each. Silicon is one of the most abundant materials on the earth—it is an ingredient in common sand. Silicon-chip technology has made computing so economical that hundreds of millions of general-purpose computers are in use worldwide, helping people in business, industry, government and their personal lives. Given the current rate of technological development, this number could easily double over the next few years.

In beginning to study this text, you are starting on a challenging and rewarding educational path. As you proceed, if you would like to communicate with us, please send us e-mail at **`deitel@deitel.com`** or browse our World Wide Web sites at **`www.deitel.com`**, **`www.prenhall.com/deitel`** and **`www.InformIT.com/deitel`**. We hope you enjoy learning Python with *Python How to Program*.

1.2 What Is a Computer?

A *computer* is a device capable of performing computations and making logical decisions at speeds millions and even billions of times faster than those of human beings. For example, many of today's personal computers can perform hundreds of millions—even billions—of additions per second. A person operating a desk calculator might require decades to complete the same number of calculations that a powerful personal computer can perform in one second. (*Points to ponder*: How would you know whether the person added the numbers correctly? How would you know whether the computer added the numbers correctly?) Today's fastest *supercomputers* can perform hundreds of billions of additions per second—about as many calculations as hundreds of thousands of people could perform in one year! Trillion-instruction-per-second computers are already functioning in research laboratories!

Computers process *data* under the control of sets of instructions called *computer programs.* These programs guide computers through orderly sets of actions that are specified by individuals known as *computer programmers.*

A computer is composed of various devices (such as the keyboard, screen, mouse, disks, memory, CD-ROM and processing units) known as *hardware.* The programs that run on a computer are referred to as *software.* Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. Software-devel-

opment costs, however, have been rising steadily, as programmers develop ever more powerful and complex applications without being able to improve significantly the technology of software development. In this book, you will learn proven software-development methods that can reduce software-development costs—top-down, stepwise refinement, functionalization and object-oriented programming. Object-oriented programming is widely believed to be the significant breakthrough that can greatly enhance programmer productivity.

1.3 Computer Organization

Virtually every computer, regardless of differences in physical appearance, can be envisioned as being divided into six *logical units*, or sections:

1. *Input unit.* This "receiving" section of the computer obtains information (data and computer programs) from various *input devices*. The input unit then places this information at the disposal of the other units to facilitate the processing of the information. Today, most users enter information into computers via keyboards and mouse devices. Other input devices include microphones (for speaking to the computer), scanners (for scanning images) and digital cameras and video cameras (for taking photographs and making videos).
2. *Output unit.* This "shipping" section of the computer takes information that the computer has processed and places it on various *output devices*, making the information available for use outside the computer. Computers can output information in various ways, including displaying the output on screens, playing it on audio/video devices, printing it on paper or using the output to control other devices.
3. *Memory unit.* This is the rapid-access, relatively low-capacity "warehouse" section of the computer, which facilitates the temporary storage of data. The memory unit retains information that has been entered through the input unit, enabling that information to be immediately available for processing. In addition, the unit retains processed information until that information can be transmitted to output devices. Often, the memory unit is called either *memory* or *primary memory*—*random access memory* (*RAM*) is an example of primary memory. Primary memory is usually volatile, which means that it is erased when the machine is powered off.
4. *Arithmetic and logic unit* (*ALU*). The ALU is the "manufacturing" section of the computer. It is responsible for the performance of calculations such as addition, subtraction, multiplication and division. It also contains decision mechanisms, allowing the computer to perform such tasks as determining whether two items stored in memory are equal.
5. *Central processing unit* (*CPU*). The CPU serves as the "administrative" section of the computer. This is the computer's coordinator, responsible for supervising the operation of the other sections. The CPU alerts the input unit when information should be read into the memory unit, instructs the ALU about when to use information from the memory unit in calculations and tells the output unit when to send information from the memory unit to certain output devices.
6. *Secondary storage unit.* This unit is the long-term, high-capacity "warehousing" section of the computer. Secondary storage devices, such as hard drives and disks,

normally hold programs or data that other units are not actively using; the computer then can retrieve this information when it is needed—hours, days, months or even years later. Information in secondary storage takes much longer to access than does information in primary memory. However, the price per unit of secondary storage is much less than the price per unit of primary memory. Secondary storage is usually *nonvolatile*—it retains information even when the computer is off.

1.4 Evolution of Operating Systems

Early computers were capable of performing only one *job* or *task* at a time. In this mode of computer operation, often called single-user *batch processing*, the computer runs one program at a time and processes data in groups called *batches*. Users of these early systems typically submitted their jobs to a computer center on decks of punched cards. Often, hours or even days elapsed before results were returned to the users' desks.

To make computer use more convenient, software systems called *operating systems* were developed. Early operating systems oversaw and managed computers' transitions between jobs. By minimizing the time it took for a computer operator to switch from one job to another, the operating system increased the total amount of work, or *throughput*, computers could process in a given time period.

As computers became more powerful, single-user batch processing became inefficient, because computers spent a great deal of time waiting for slow input/output devices to complete their tasks. Developers then looked to multiprogramming techniques, which enabled many tasks to *share* the resources of the computer to achieve better utilization. *Multiprogramming* involves the "simultaneous" operation of many jobs on a computer that splits its resources among those jobs. However, users of early multiprogramming operating systems still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several industry and university groups pioneered *timesharing* operating systems. Timesharing is a special type of multiprogramming that allows users to access a computer through *terminals* (devices with keyboards and screens). Dozens or even hundreds of people can use a timesharing computer system at once. It is important to note that the computer does not actually run all the users' requests simultaneously. Rather, it performs a small portion of one user's job and moves on to service the next user. However, because the computer does this so quickly, it can provide service to each user several times per second. This gives users' programs the appearance of running simultaneously. Timesharing offers major advantages over previous computing systems in that users receive prompt responses to requests, instead of waiting long periods to obtain results.

The UNIX operating system, which is now widely used for advanced computing, originated as an experimental timesharing operating system. Dennis Ritchie and Ken Thompson developed UNIX at Bell Laboratories beginning in the late 1960s and developed C as the programming language in which they wrote it. They freely distributed the source code to other programmers who wanted to use, modify and extend it. A large community of UNIX users quickly developed. The operating system and the world of the C language grew as UNIX users contributed their own programs and tools. Through a collaborative effort among numerous researchers and developers, UNIX became a powerful and flexible operating system able to handle almost any type of task that a user required. Many versions of UNIX have evolved, including today's phenomenally popular, *open-source*, Linux operating system.

1.5 Personal Computing, Distributed Computing and Client/Server Computing

In 1977, Apple Computer popularized the phenomenon of *personal computing.* Initially, it was a hobbyist's dream. However, the price of computers soon dropped so far that large numbers of people could buy them for personal or business use. In 1981, IBM, the world's largest computer vendor, introduced the IBM Personal Computer. Personal computing rapidly became legitimate in business, industry and government organizations.

The computers first pioneered by Apple and IBM were "stand-alone" units—people did their work on their own machines and transported disks back and forth to share information. (This process was often called "sneakernet.") Although early personal computers were not powerful enough to timeshare several users, the machines could be linked together into computer networks, either over telephone lines or via *local area networks* (*LANs*) within an organization. These networks led to the *distributed computing* phenomenon, in which an organization's computing is distributed over networks to the sites at which the work of the organization is performed, instead of being performed only at a central computer installation. Personal computers were powerful enough to handle both the computing requirements of individual users and the basic tasks involved in the electronic transfer of information between computers. *N-tier applications* split up an application over numerous distributed computers. For example, a *three-tier application* might have a user interface on one computer, business-logic processing on a second and a database on a third; all interact as the application runs.

Today's most advanced personal computers are as powerful as the million-dollar machines of just two decades ago. High-powered desktop machines—called *workstations*—provide individual users with enormous capabilities. Information is easily shared across computer networks, in which computers called *servers* store programs and data that can be used by *client* computers distributed throughout the network. This type of configuration gave rise to the term *client/server computing.* Today's popular operating systems, such as UNIX, Solaris, MacOS, Windows 2000, Windows XP and Linux, provide the kinds of capabilities discussed in this section.

1.6 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others that require intermediate *translation* steps. Although hundreds of computer languages are in use today, the diverse offerings can be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Any computer can understand only its own *machine language* directly. As the "natural language" of a particular computer, machine language is defined by the computer's hardware design. Machine languages generally consist of streams of numbers (ultimately reduced to 1s and 0s) that instruct computers how to perform their most elementary operations. Machine languages are *machine-dependent*, which means that a particular machine language can be used on only one type of computer. The following section of a machine-

language program, which adds *overtime* pay to *base pay* and stores the result in *gross pay*, demonstrates the incomprehensibility of machine language to the human reader.

```
+1300042774
+1400593419
+1200274027
```

As the popularity of computers increased, machine-language programming proved to be excessively slow, tedious and error prone. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent the elementary operations of the computer. These abbreviations formed the basis of *assembly languages*. *Translator programs* called *assemblers* convert assembly language programs to machine language at computer speeds. The following section of an assembly-language program also adds *overtime pay* to *base pay* and stores the result in *gross pay*, but presents the steps more clearly to human readers than does its machine-language equivalent:

```
LOAD   BASEPAY
ADD    OVERPAY
STORE  GROSSPAY
```

Such code is clearer to humans but incomprehensible to computers until translated into machine language.

Although computer use increased rapidly with the advent of assembly languages, these languages still required many instructions to accomplish even the simplest tasks. To speed up the programming process, *high-level languages*, in which single statements accomplish substantial tasks, were developed. Translation programs called *compilers* convert high-level-language programs into machine language. High-level languages enable programmers to write instructions that look almost like everyday English and contain common mathematical notations. A payroll program written in a high-level language might contain a statement such as

```
grossPay = basePay + overTimePay
```

Obviously, programmers prefer high-level languages to either machine languages or assembly languages. C, C++, C# (pronounced “C sharp”), Java, Visual Basic, Perl and Python are among the most popular high-level languages.

Compiling a high-level language program into machine language can require a considerable amount of time. This problem was solved by the development of *interpreter* programs that can execute high-level language programs directly, bypassing the compilation step, and interpreters can start running a program immediately without “suffering” a compilation delay. Although programs that are already compiled execute faster than interpreted programs, interpreters are popular in program-development environments. In these environments, developers change programs frequently as they add new features and correct errors. Once a program is fully developed, a compiled version can be produced so that the program runs at maximum efficiency. As we will see throughout this book, interpreted languages—like Python—are particularly popular for implementing World Wide Web applications.

1.7 Structured Programming

During the 1960s, many large software-development efforts encountered severe difficulties. Development typically ran behind schedule, costs often greatly exceeded budgets and

the finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity, intended to address these issues, resulted in the evolution of *structured programming*—a disciplined approach to the creation of programs that are clear, demonstrably correct and easy to modify.

One of the more tangible results of this research was the development of the *Pascal* programming language in 1971. Pascal, named after the seventeenth-century mathematician and philosopher Blaise Pascal, was designed for teaching structured programming in academic environments and rapidly became the preferred introductory programming language in most universities. Unfortunately, because the language lacked many features needed to make it useful in commercial, industrial and government applications, it was not widely accepted in these environments. By contrast, C, which also arose from research on structured programming, did not have the limitations of Pascal, and became extremely popular.

The *Ada* programming language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. Hundreds of programming languages were being used to produce DOD's massive command-and-control software systems. DOD wanted a single language that would meet its needs. Pascal was chosen as a base, but the final Ada language is quite different from Pascal. The language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is generally credited with writing the world's first computer program, in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada is *multitasking*, which allows programmers to specify that many activities are to occur in parallel. As we will see in Chapters 18–19, Python offers process management and *multithreading*—two capabilities that enable programs to specify that various activities are to proceed in parallel.

1.8 Object-Oriented Programming

One of the authors, HMD, remembers the great frustration felt in the 1960s by software-development organizations, especially those developing large-scale projects. During the summers of his undergraduate years, HMD had the privilege of working at a leading computer vendor on the teams developing time-sharing, virtual-memory operating systems. It was a great experience for a college student, but, in the summer of 1967, reality set in. The company "decommitted" from producing as a commercial product the particular system that hundreds of people had been working on for several years. It was difficult to get this software right. Software is "complex stuff."

As the benefits of structured programming (and the related disciplines of *structured systems analysis and design*) were realized in the 1970s, improved software technology did begin to appear. However, it was not until the technology of object-oriented programming became widely used in the 1980s and 1990s that software developers finally felt they had the necessary tools to improve the software-development process dramatically.

Actually, object technology dates back to at least the mid-1960s, but no broad-based programming language incorporated the technology until C++. Although not strictly an object-oriented language, C++ absorbed the capabilities of C and incorporated Simula's ability to create and manipulate objects. C++ was never intended for widespread use beyond the research laboratories at AT&T, but grass-roots support rapidly developed for the hybrid language.

What are objects, and why are they special? Object technology is a packaging scheme that facilitates the creation of meaningful software units. These units are large and focused on particular applications areas. There are date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects and so on. In fact, almost any noun can be reasonably represented as a software object. Objects have *properties* (i.e., *attributes*, such as color, size and weight) and perform *actions* (i.e., *behaviors*, such as moving, sleeping or drawing). Classes represent groups of related objects. For example, all cars belong to the "car" class, even though individual cars vary in make, model, color and options packages. A class specifies the general format of its objects; the properties and actions available to an object depend on its class.

We live in a world of objects. Just look around you—there are cars, planes, people, animals, buildings, traffic lights, elevators and so on. Before object-oriented languages appeared, *procedural programming languages* (such as Fortran, Pascal, BASIC and C) focused on actions (verbs) rather than things or objects (nouns). We live in a world of objects, but earlier programming languages forced individuals to program primarily with verbs. This paradigm shift made program writing a bit awkward. However, with the advent of popular object-oriented languages, such as C++, Java, C# and Python, programmers can program in an object-oriented manner that reflects the way in which they perceive the world. This process, which seems more natural than procedural programming, has resulted in significant productivity gains.

One of the key problems with procedural programming is that the program units created do not mirror real-world entities effectively and therefore are not particularly reusable. Programmers often write and rewrite similar software for various projects. This wastes precious time and money as people repeatedly "reinvent the wheel." With object technology, properly designed software entities (called objects) can be reused on future projects. Using libraries of reusable componentry can greatly reduce the amount of effort required to implement certain kinds of systems (as compared to the effort that would be required to reinvent these capabilities in new projects).

Some organizations report that software reusability is not, in fact, the key benefit of object-oriented programming. Rather, they indicate that object-oriented programming tends to produce software that is more understandable because it is better organized and has fewer maintenance requirements. As much as 80 percent of software costs are not associated with the original efforts to develop the software, but instead are related to the continued evolution and maintenance of that software throughout its lifetime. Object orientation allows programmers to abstract the details of software and focus on the "big picture." Rather than worrying about minute details, the programmer can focus on the behaviors and interactions of objects. A roadmap that showed every tree, house and driveway would be difficult, if not impossible, to read. When such details are removed and only the essential information (roads) remains, the map becomes easier to understand. In the same way, a program that is divided into objects is easy to understand, modify and update because it hides much of the detail. It is clear that object-oriented programming will be the key programming methodology for at least the next decade.

1.9 Hardware Trends

Every year, people generally expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especial-

ly with regard to the costs of hardware supporting these technologies. For many decades, and continuing into the foreseeable future, hardware costs have fallen rapidly, if not precipitously. Every year or two, the capacities of computers approximately double.[1] This is especially true in relation to the amount of memory that computers have for programs, the amount of secondary storage (such as disk storage) computers have to hold programs and data over longer periods of time and their processor speeds—the speeds at which computers execute their programs (i.e., do their work). Similar improvements have occurred in the communications field, in which costs have plummeted as enormous demand for *bandwidth* (i.e., information-carrying capacity of communication lines) has attracted tremendous competition. We know of no other fields in which technology moves so quickly and costs fall so rapidly. Such phenomenal improvement in the computing and communications fields is truly fostering the so-called *Information Revolution*.

When computer use exploded in the 1960s and 1970s, many people discussed the dramatic improvements in human productivity that computing and communications would cause. However, these improvements did not materialize. Organizations were spending vast sums of capital on computers and employing them effectively, but without fully realizing the expected productivity gains. The invention of microprocessor chip technology and its wide deployment in the late 1970s and 1980s laid the groundwork for the productivity improvements that individuals and businesses have achieved in recent years.

1.10 History of the Internet and World Wide Web

In the late 1960s, one of the authors (HMD) was a graduate student at MIT. His research at MIT's Project Mac (now the Laboratory for Computer Science—the home of the World Wide Web Consortium) was funded by ARPA—the Advanced Research Projects Agency of the Department of Defense. ARPA sponsored a conference at which several dozen ARPA-funded graduate students were brought together at the University of Illinois at Urbana-Champaign to meet and share ideas. During this conference, ARPA rolled out the blueprints for networking the main computer systems of approximately a dozen ARPA-funded universities and research institutions. The computers were to be connected with communications lines operating at a then-stunning 56 Kbps (1 Kbps is equal to 1,024 bits per second), at a time when most people (of the few who had access to networking technologies) were connecting over telephone lines to computers at a rate of 110 bits per second. HMD vividly recalls the excitement at that conference. Researchers at Harvard talked about communicating with the Univac 1108 "supercomputer," which was located across the country at the University of Utah, to handle calculations related to their computer graphics research. Many other intriguing possibilities were discussed. Academic research was about to take a giant leap forward. Shortly after this conference, ARPA proceeded to implement what quickly became called the *ARPAnet*, the grandparent of today's *Internet.*

Things worked out differently from the original plan. Although the ARPAnet did enable researchers to network their computers, its chief benefit proved to be the capability for quick and easy communication via what came to be known as *electronic mail* (*e-mail*). This is true even on today's Internet, with e-mail, instant messaging and file transfer facilitating communications among hundreds of millions of people worldwide.

1. This often is called *Moore's Law.*

The network was designed to operate without centralized control. This meant that, if a portion of the network should fail, the remaining working portions would still be able to route data packets from senders to receivers over alternative paths.

The protocol (i.e., set of rules) for communicating over the ARPAnet became known as the *Transmission Control Protocol (TCP)*. TCP ensured that messages were properly routed from sender to receiver and that those messages arrived intact.

In parallel with the early evolution of the Internet, organizations worldwide were implementing their own networks to facilitate both intra-organization (i.e., within the organization) and inter-organization (i.e., between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to enable these diverse products to communicate with each other. ARPA accomplished this by developing the *Internet Protocol* (*IP),* which created a true "network of networks," the current architecture of the Internet. The combined set of protocols is now commonly called *TCP/IP*.

Initially, use of the Internet was limited to universities and research institutions; later, the military adopted the technology. Eventually, the government decided to allow access to the Internet for commercial purposes. When this decision was made, there was resentment among the research and military communities—it was felt that response times would become poor as "the Net" became saturated with so many users.

In fact, the opposite has occurred. Businesses rapidly realized that, by making effective use of the Internet, they could refine their operations and offer new and better services to their clients. Companies started spending vast amounts of money to develop and enhance their Internet presence. This generated fierce competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. The result is that bandwidth on the Internet has increased tremendously, while hardware costs have plummeted. It is widely believed that the Internet played a significant role in the economic growth that many industrialized nations experienced over the last decade.

The *World Wide Web (WWW)* allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animations, audios and/or videos) on almost any subject. Even though the Internet was developed more than three decades ago, the introduction of the World Wide Web was a relatively recent event. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began to develop a technology for sharing information via hyperlinked text documents. Basing the new language on the well-established *Standard Generalized Markup Language* (*SGML*)—a standard for business data interchange—Berners-Lee called his invention the *HyperText Markup Language* (*HTML*). He also wrote communication protocols to form the backbone of his new hypertext information system, which he referred to as the World Wide Web.

Historians will surely list the Internet and the World Wide Web among the most important and profound creations of humankind. In the past, most computer applications ran on "stand-alone" computers (computers that were not connected to one another). Today's applications can be written to communicate among the world's hundreds of millions of computers. The Internet and World Wide Web merge computing and communications technologies, expediting and simplifying our work. They make information instantly and conveniently accessible to large numbers of people. They enable individuals and small businesses to achieve worldwide exposure. They are profoundly changing the way we do business and conduct our personal lives. People can search for the best prices on virtually

any product or service. Special-interest communities can stay in touch with one another. Researchers can be made instantly aware of the latest breakthroughs worldwide.

We have written two books for academic courses that convey fundamental principles of computing in the context of Internet and World Wide Web programming—*Internet and World Wide Web How to Program: Second Edition* and *e-Business and e-Commerce How to Program*.

1.11 World Wide Web Consortium (W3C)

In October 1994, Tim Berners-Lee founded an organization, called the *World Wide Web Consortium* (*W3C*), that is devoted to developing nonproprietary, interoperable technologies for the World Wide Web. One of the W3C's primary goals is to make the Web universally accessible—regardless of disabilities, language or culture.

The W3C is also a standardization organization and is comprised of three *hosts*—the Massachusetts Institute of Technology (MIT), France's INRIA (Institut National de Recherche en Informatique et Automatique) and Keio University of Japan—and over 400 members, including Deitel & Associates, Inc. Members provide the primary financing for the W3C and help provide the strategic direction of the Consortium. To learn more about the W3C, visit **www.w3.org**.

Web technologies standardized by the W3C are called *Recommendations*. Current W3C Recommendations include *Extensible HyperText Markup Language (XHTML™)*, *Cascading Style Sheets (CSS™)* and the *Extensible Markup Language (XML)*. Recommendations are not actual software products, but documents that specify the role, syntax and rules of a technology. Before becoming a W3C Recommendation, a document passes through three major phases: *Working Draft*—which, as its name implies, specifies an evolving draft; *Candidate Recommendation*—a stable version of the document that industry can begin to implement; and *Proposed Recommendation*—a Candidate Recommendation that is considered mature (i.e., has been implemented and tested over a period of time) and is ready to be considered for W3C Recommendation status. For detailed information about the W3C Recommendation track, see "6.2 The W3C Recommendation track" at

```
www.w3.org/Consortium/Process/Process-19991111/
process.html#RecsCR
```

1.12 Extensible Markup Language (XML)

As the popularity of the Web exploded, HTML's limitations became apparent. HTML's lack of *extensibility* (the ability to change or add features) frustrated developers, and its ambiguous definition allowed erroneous HTML to proliferate. In response to these problems, the W3C added limited extensibility to HTML. This was, however, only a temporary solution—the need for a standardized, fully extensible and structurally strict language was apparent. As a result, XML was developed by the W3C. XML combines the power and extensibility of its parent language, Standard Generalized Markup Language (SGML), with the simplicity that the Web community demands. At the same time, the W3C began developing XML-based standards for style sheets and advanced hyperlinking. *Extensible Stylesheet Language* (*XSL*) incorporates elements of both Cascading Style Sheets (CSS), which is used to format HTML documents and *Document Style and Semantics Specification Language* (*DSSSL*), which is used to format SGML documents. Similarly, the *Exten-*

sible Linking Language (*XLink*) combines ideas from *HyTime* and the *Text Encoding Initiative* (*TEI*), to provide extensible linking of resources.

Data independence, the separation of content from its presentation, is the essential characteristic of XML. Because an XML document describes data, any application conceivably can process an XML document. Recognizing this, software developers are integrating XML into their applications to improve Web functionality and interoperability. XML's flexibility and power make it perfect for the middle tier of client/server systems, which must interact with a wide variety of clients. Much of the processing that was once limited to server computers now can be performed by client computers, because XML's semantic and structural information enables it to be manipulated by any application that can process text. This reduces server loads and network traffic, resulting in a faster, more efficient Web.

XML is not limited to Web applications. Increasingly, XML is being employed in databases—the structure of an XML document enables it to be integrated easily with database applications. As applications become more Web enabled, it seems likely that XML will become the universal technology for data representation. All applications employing XML would be able to communicate, provided that they could understand each other's XML markup, or *vocabulary*.

Simple Object Access Protocol (SOAP) is a technology for the distribution of objects (marked up as XML) over the Internet. Developed primarily by Microsoft and DevelopMentor, SOAP provides a framework for expressing application semantics, encoding that data and packaging it in modules. SOAP has three parts: The *envelope*, which describes the content and intended recipient of a SOAP message; the SOAP *encoding rules*, which are XML-based; and the SOAP *Remote Procedure Call* (*RPC*) *representation* for commanding other computers to perform a task. SOAP is supported by many platforms, because of its foundations in XML and HTTP. We discuss XML in Chapter 15, Extensible Markup Language (XML) and in Chapter 16, XML Processing.

1.13 Open-Source Software Revolution

When the source code of a program is freely available to any developer to modify, to redistribute and to use as a basis for other software, it is called *open-source software*.[2] In contrast, *closed-source software* restricts other developers from creating software programs whose source code is based on closed-source programs.

The concept of open-source technologies is not new. The development of open-source technologies was an important factor in the growth of modern computing in 1960s. Specifically, the United States government funded what became today's Internet and encouraged computer scientists to develop technologies that could facilitate distributed computing on various computer platforms. [3] Out of these efforts came technologies such as the protocols used to communicate over today's Internet. After the Internet was established, closed-source technologies and software became the norm in the software industry, and open-source fell from popular use in the 1980s and early 1990s. In response to the "closed"

2. The Open Source Initiative's definition includes nine requirements to which software must comply before it is considered "open source." To view the entire definition, visit **<www.opensource.org/docs/definition.html>**.
3. **<www.opensource.org>**.

nature of most commercial software and programmers' frustrations with the lack of responsiveness from closed-source vendors, open-source software, regained popularity. Today, Python is part of a growing open-source software community, which includes the Linux operating system, the Perl scripting language, the Apache Web server and hundreds of other software projects.

Some people in the computer industry equate open-source with "free" software. In most cases, this is true. However, "free" in the context of open-source software is thought of most appropriately as "freedom"—the freedom for any developer to modify source code, to exchanges ideas, to participate in the software-development process and to develop new software programs based on existing open-source software. Most open-source software is copyrighted and licenses are associated with the use of the software. Open-source licenses vary in their terms; some impose few restrictions (e.g., the Artistic license[4]), whereas others require many restrictions on the manner in which the software may be modified and used. Usually, either an individual developer or an organization maintains the software copyrights. To view an example of a license, visit **`www.python.org/2.2/license.html`** to read the Python agreement.

Typically, the source code for open-source products is available for download over the Internet. This enables developers to learn from, validate and modify the source code to meet their own needs. With a community of developers, more people review the code so issues such as performance and security problems are detected and resolved faster than they would be in closed-source software development. Additionally, a larger community of developers can contribute more features. Often, code fixes are available within hours, and new versions of open-source software are available more frequently than are versions of closed-source software. Open-source licenses often require that developers publish any enhancements they make so that the open-source community can continue to evolve those products. For example, Python developers participate in the **`comp.lang.python`** newsgroup to exchange ideas regarding the development of Python. Python developers also can document and submit their modifications to the Python Software Foundation through Python Enhancement Proposals (PEPS), which enables the Python group to evaluate the proposed changes and incorporate the ones they choose in future releases.[5]

Many companies, (e.g., IBM, Red Hat and Sun) support open-source developers and projects. Sometimes companies take open-source applications and sell them commercially (this depends on software licensing). For-profit companies also provide services such as support, custom-made software and training. Developers can offer their services as consultants or trainers to businesses implementing the software.[6] For more information about open-source software, visit the Open Source Initiative's Web site at **`www.opensource.org`**.

1.14 History of Python

Python began in late 1989. At that time, Guido van Rossum, a researcher at the *National Research Institute for Mathematics and Computer Science in Amsterdam (CWI)*, needed a high-level scripting language to accomplish administrative tasks for his research group's

4. **`<www.opensource.org/licenses/artistic-license.html>`**.
5. **`<www.python.org>`**.
6. **`<www-106.ibm.com/developerworks/opensource/library/license.html?dwzone=opensource>`**.

Amoeba distributed operating system. To create this new language, he drew heavily from *All Basic Code (ABC)*—a high-level teaching language—for syntax, and from *Modula-3*, a systems programming language, for error-handling techniques. However, one major shortcoming of ABC was its lack of extensibility; the language was not open to improvements or extensions. So, van Rossum decided to create a language that combined many of the elements he liked from existing languages, but one that could be extended through classes and programming interfaces. He named this language Python, after the popular comic troupe Monty Python.

Since its public release in early 1991, a growing community of Python developers and users have improved it to create a mature and well-supported programming language. Python has been used to develop a variety of applications, from creating online e-mail programs to controlling underwater vehicles, configuring operating systems and creating animated films. In 2001, the core Python development team moved to Digital Creations, the creators of *Zope*—a Web application server written in Python. It is expected that Python will continue to grow and expand into new programming realms.

1.15 Python Modules

Python is a modularly extensible language; it can incorporate new *modules* (reusable pieces of software). These new modules, which can be written by any Python developer, extend Python's capabilities. The primary distribution center for Python source code, modules and documentation is the Python Web site—**`www.python.org`**—with plans to develop a site dedicated solely to maintaining Python modules.

1.16 General Notes about Python and This Book

Python was designed so that novice and experienced programmers could learn and understand the language quickly and use it with ease. Unlike its predecessors, Python was designed to be portable and extensible. Python's syntax and design promote good programming practices and tend to produce surprisingly rapid development times without sacrificing program scalability and maintenance.

Python is simple enough to be used by beginning programmers, but powerful enough to attract professionals. *Python How to Program* introduces programming concepts through abundant, complete, working examples and discussions. As we progress, we begin to explore more complex topics by creating practical applications. Throughout the book, we emphasize good programming practices and portability tips and explain how to avoid common programming errors.

Python is one of the most highly portable programming languages in existence. Originally, it was implemented on UNIX, but has since spread to many other platforms, including Microsoft Windows and Apple Mac OS X. Python programs often can be ported from one operating system to another without any change and still execute properly.

1.17 Tour of the Book

In this section, we take a tour of the subjects introduced in *Python How to Program*. Some chapters end with an Internet and World Wide Web Resources section, which lists resources that provide additional information on Python programming.

Chapter 1—Introduction to Computers, the Internet and the World Wide Web
In this chapter, we discuss what computers are, how they work and how they are programmed. The chapter introduces structured programming and explains why this set of techniques has fostered a revolution in the way programs are written. A brief history of the development of programming languages—from machine languages, to assembly languages to high-level languages—is included. We present some historical information about computers and computer programming and introductory information about the Internet and the World Wide Web. We discuss the origins of the Python programming language and overview the concepts introduced in the remaining chapters of the book.

Chapter 2—Introduction to Python Programming
Chapter 2 introduces a typical Python programming environment and the basic syntax for writing Python programs. We discuss how to run Python from the command line. In addition to the interpreter, Python can execute statements in an interactive mode in which Python statements can be typed and executed. Throughout the chapter and the book, we include several interactive sessions to highlight and illustrate various subtle programming points. In this chapter, we discuss variables and introduce arithmetic, assignment, equality, relational and string operators. We introduce decision-making and arithmetic operations. Strings are a basic and powerful built-in data type. We introduce some standard output-formatting techniques. We discuss the concept of *objects* and *variables*. Objects are containers for values and variables are names that reference objects. Our Python programs use syntax coloring to highlight keywords, comments and regular program text. After studying this chapter, readers will understand how to write simple but complete Python programs.

Chapter 3—Control Structures
This chapter introduces *algorithms* (procedures) for solving problems. It explains the importance of using control structures effectively in producing programs that are understandable, debuggable, maintainable and more likely to work properly on the first try. The chapter introduces selection structures (**`if`**, **`if`**/**`else`** and **`if`**/**`elif`**/**`else`**) and repetition structures (**`while`** and **`for`**). It examines repetition in detail and compares counter-controlled and sentinel-controlled loops. We explain the technique of top-down, stepwise refinement which is critical to the production of properly structured programs and the creation of the popular program design aid, *pseudocode*. The chapter examples and case studies demonstrate how quickly and easily pseudocode algorithms can be converted to working Python code. The chapter contains an explanation of **`break`** and **`continue`**—statements that alter the flow of control. We show how to use the logical operators **`and`**, **`or`** and **`not`** to enable programs to make sophisticated decisions. The chapter includes several interactive sessions that demonstrate how to create a **`for`** structure and how to avoid several common programming errors that arise in structured programming. The chapter concludes with a summary of structured programming. The techniques presented in Chapter 3 are applicable for effective use of control structures in any programming language, not just Python. This chapter helps the student develop good programming habits in preparation for dealing with the more substantial programming tasks in the remainder of the text.

Chapter 4—Functions
Chapter 4 discusses the design and construction of *functions*. Python's function-related capabilities include built-in functions, programmer-defined functions and recursion. The

techniques presented in Chapter 4 are essential for creating properly structured programs—especially the larger programs and software that system programmers and application programmers are likely to develop in real-world applications. The "divide and conquer" strategy is presented as an effective means for solving complex problems by dividing them into simpler interacting components. We begin by introducing modules as containers for groups of useful functions. We introduce module **`math`** and discuss the many mathematics-related functions the module contains. Students enjoy the treatment of random numbers and simulation, and they are entertained by a study of the dice game, craps, which makes elegant use of control structures. The chapter illustrates how to solve a Fibonacci and factorial problem using a programming technique called *recursion* in which a function calls itself. Scope rules are discussed in the context of an example that examines local and global variables. The chapter also discusses the various ways a program can import a module and its elements and how the **`import`** statement affects the program's *namespace*. Python functions can specify default arguments and keyword arguments. We discuss both ways of passing information to functions and illustrate some common programming errors in an interactive session. The exercises present traditional mathematics and computer-science problems, including how to solve the famous Towers of Hanoi problem using recursion. Another exercise asks the reader to display the prime numbers from 2–100.

Chapter 5—Lists, Tuples and Dictionaries

This chapter presents a detailed introduction to three high-level Python data types: *lists*, *tuples* and *dictionaries*. These data types enable Python programmers to accomplish complex tasks through minimal lines of code. Strings, lists and tuples are all *sequences*—a data type that can be manipulated through indexing and "slicing." We discuss how to create, access and manipulate sequences and present an example that creates a histogram from a sequence of values. We consider the different ways lists and tuples are used in Python programs. Dictionaries are "mappable" types—keys are stored with (or mapped to) their associated values. We discuss how to create, initialize and manipulate dictionaries in an example that stores student grades. We introduce *methods*—functions that perform the operations of objects, such as lists and dictionaries—and how to use methods to access, sort and search data. These methods easily perform algorithmic tasks that normally require abundant lines of code in other languages. We consider immutable sequences—which cannot be altered—and mutable sequences—which can be altered. An important and perhaps unexpected "side effect" occurs when passing mutable sequences to functions—we present an example to show the ramifications of this side effect. The exercises at the end of the chapter address elementary sorting and searching algorithms and other programming techniques.

Chapter 6—Introduction to the Common Gateway Interface (CGI)

Chapter 6 illustrates a protocol for interactions between applications (CGI programs or scripts) and Web servers. The chapter introduces the *HyperText Transfer Protocol (HTTP)*, which is a fundamental component in the communication of data between a Web server and a Web browser. We explain how a client computer connects to a server computer to request information over the Internet and how a Web server runs a CGI program then sends a response to the client. The most common data sent from a Web server to a Web browser is a Web page—a document that is formatted with the *Extensible HyperText Markup Language (XHTML)*. In this chapter, we learn how to create simple CGI scripts. We also show how to send user input from a browser to a CGI script with an example that displays a person's

name in a Web browser. We then focus on how to send user input to a CGI script by using an XHTML form to pass data between the client and the CGI program on the server. We demonstrate how to use module **cgi** to process form data. The chapter contains descriptions of various HTTP headers used with CGI. We conclude by integrating the CGI material into a Web portal case study that allows the user to log in to a fictional travel Web site and to view information about special offers.

Chapter 7—Object-Based Programming

In this chapter, we begin our discussion of object-based programming. The chapter represents a wonderful opportunity for teaching *data abstraction* the "right way"—through the Python language that was designed from the ground up to be object-oriented. In recent years, data abstraction has become an important topic in introductory computing courses. We discuss how to implement a time abstract data type with a class and how to initialize and access data members of the class. Unlike other languages, Python does not permit programmers to prohibit attribute access. In this and the next two chapters, we discuss several access-control techniques. We introduce "private" attributes as well as *get* and *set* methods that control access to data. All objects and classes have attributes in common, and we discuss their names and values. We discuss default constructors and expand our example further. We also introduce the **raise** statement for indicating errors. Classes can contain class attributes—data that are created once and used by all instances of the class. We also discuss an example of composition, in which instances contain references to other instances as data members. The chapter concludes with a discussion of software reusability. The more mathematically inclined reader will enjoy the exercise on creating class **Rational** (for rational numbers).

Chapter 8—Customizing Classes

This chapter discusses the several methods Python provides for customizing the behavior of a class. These methods extend the access-control mechanism introduced in the previous chapter. Perhaps the most powerful of the customization techniques is operator overloading, which enables the programmer to tell the Python interpreter how to use existing operators with objects of new types. Python already knows how to use these operators with objects of built-in types such as integers, lists and strings. But suppose we create a new **Rational** class—what would the plus sign (**+**) denote when used between **Rational** objects? In this chapter, the programmer will learn how to "overload" the plus sign so that, when it is written between two **Rational** objects in an expression, the interpreter will generate a method call to an "operator method" that "adds" the two **Rational** objects. The chapter discusses the fundamentals of operator overloading, restrictions in operator overloading, overloading unary and binary operators and converting between types. The chapter also discusses how to customize a class so it contains list- or dictionary-like behaviors. The more mathematically inclined student will enjoy creating class **Polynomial**.

Chapter 9—Object-Oriented Programming: Inheritance

This chapter introduces one of the most fundamental capabilities of object-oriented programming languages: *inheritance*. Inheritance is a form of software reusability in which new classes are developed quickly and easily by absorbing the capabilities of existing classes and adding appropriate new capabilities. The chapter discusses the notions of base classes and derived classes, direct-base classes, indirect-base classes, constructors and

destructors in base classes and derived classes, and software engineering with inheritance. This chapter compares various object-oriented relationships, such as inheritance and composition. Inheritance leads to programming techniques that highlight one of Python's most powerful built-in features—*polymorphism*. When many classes are related through inheritance to a common base class, each derived-class object may be treated as a base-class instance. This enables programs to be written in a general manner independent of the specific types of the derived-class objects. New kinds of objects can be handled by the same program, thus making systems more extensible. This style of programming is commonly used to implement today's popular *graphical user interfaces (GUIs)*. The chapter concludes with a discussion of the new object-oriented programming techniques available in Python version 2.2.

Chapter 10—Graphical User Interface Components: Part 1

Chapter 10 introduces **`Tkinter`**, a module that provides a Python interface to the popular *Tool Command Language/Tool Kit (Tcl/Tk)* graphical-user-interface (GUI) toolkit. The chapter begins with a detailed overview of the **`Tkinter`** module. Using **`Tkinter`**, the programmer can create graphical programs quickly and easily. We illustrate several basic **`Tkinter`** *components*—**`Label`**, **`Button`**, **`Entry`**, **`Checkbutton`** and **`Radiobutton`**. We discuss the concept of event-handling that is central to GUI programming and present examples that show how to handle mouse and keyboard events in GUI applications. We conclude the chapter with a more in-depth examination of the **`pack`**, **`grid`** and **`place`** Tk layout managers. The exercises ask the reader to use the concepts presented in the chapter to create practical applications, such as a program that allows the user to convert temperature values between scales. Another exercise asks the reader to create a GUI calculator. After completing this chapter, the reader should be able to understand most **`Tkinter`** applications.

Chapter 11—Graphical User Interface Components: Part 2

Chapter 11 discusses additional GUI-programming topics. We introduce module **`Pmw`**, which extends the basic Tk GUI widget set. We show how to create menus, popup menus, scrolled text boxes and windows. The examples demonstrate copying text from one window to another, allowing the user to select and display images, changing the text font and changing the background color of a window. Of particular interest is the 35-line program that allows the user to draw pictures on a **`Canvas`** component with a mouse. The chapter concludes with a discussion of alternative GUI toolkits available to the Python programmer, including **`pyGTK`**, **`pyOpenGL`** and **`wxWindows`**. One of the chapter exercises asks the reader to enhance the temperature-conversion example from the previous chapter. A second exercise asks the reader to create a simple program that draws a shape on the screen. In another exercise, the reader fills the shape with a color selected from menu. Many examples throughout the remainder of the book use the GUI techniques shown in Chapters 10 and 11. After completing Chapters 10 and 11, the reader will be prepared to write the GUI portions of programs that perform database operations, networking tasks and simple games.

Chapter 12—Exception Handling

This chapter enables the programmer to write programs that are more robust, more fault tolerant and more appropriate for business-critical and mission-critical environments. We be-

gin the chapter with an explanation of exception-handling techniques. We then discuss when exception handling is appropriate and introduce the basics of exception handling with **`try`**/**`except`**/**`else`** statements in an example that gracefully handles the fatal logic error of dividing by zero. The programmer can raise exceptions specifically using the **`raise`** statement; we discuss the syntax of this statement and demonstrate its use. The chapter explains how to extract information from exceptions and how and when to raise exceptions. We explain the **`finally`** statement and provide a detailed explanation of when and where exceptions are caught in programs. In Python, exceptions are classes. We discuss how exceptions relate to classes by examining the exception hierarchy and how to create custom exceptions. The chapter concludes with an example that takes advantage of the capabilities of module **`traceback`** to examine the nature and contents of Python exceptions.

Chapter 13—String Manipulation and Regular Expressions

This chapter explores how to manipulate string appearance, order and contents. Strings form the basis of most Python output. The chapter discussion includes methods **`count`**, **`find`** and **`index`**, which search strings for substrings. Method **`split`** breaks a string into a list of strings. Method **`replace`** replaces a substring of a string with another substring. These methods provide basic text manipulation capabilities, but programmers often require more powerful pattern-based text manipulation. The **`re`** regular-expression module provides pattern-based text manipulation in Python. Regular-expression processing can be a complex subject, with many pitfalls. We present several sections that range from basic regular expressions to more substantial topics. We point out the most common programming mistakes and include examples that highlight how these mistakes occur and how to avoid them. The sections discuss the common functions and classes of module **`re`** and the common regular-expression metacharacters and sequences. We demonstrate grouping, which enables programmers to retrieve information from regular-expression processing results. Python regular expressions can be compiled to improve regular-expression processing performance, so we discuss when it is appropriate to do this. The exercises ask the reader to explore common applications of regular expressions.

Chapter 14—File Processing and Serialization

In this chapter, we discuss the techniques for processing sequential-access and random-access text files. The chapter overviews the data hierarchy among bits, bytes, fields, records and files. Next, Python's simple view of files and filehandles is presented. Sequential-access files are discussed using programs that show how to open and close files, how to store data sequentially in a file and how to read data sequentially from a file. The examples use the string-formatting techniques from the previous chapter to output data read from a file. We include a more substantial program that simulates a credit-inquiry program that retrieves data from a sequential-access file and formats the output based on data obtained from the file. One feature of the chapter is the discussion of how the **`print`** statement can redirect text to an arbitrary file, including the *standard error file* to which programs display error messages. Our discussion of random-access files uses module **`shelve`**, which provides a dictionary-like interface to random-access files. We use **`shelve`** to create a file for random access and to read and write data to a **`shelve`** file. We include a larger transaction-processing programming example that employs the techniques discussed in the chapter. One benefit of Python's high-level data types and modules is that programs can serialize

(save to disk) arbitrary Python objects. We present an example that uses module **cPickle** to store a Python dictionary to disk for later use.

Chapter 15—Extensible Markup Language (XML)

XML is a language for creating markup languages. Unlike HTML, which formats information for display, XML structures information. It does not have a fixed set of tags as HTML does, but instead enables the document author to create new ones. This chapter provides a brief overview of *parsers*, which are programs that process XML documents and their data, and the requirements for a *well-formed document* (i.e., a document that is syntactically correct). We also introduce *namespaces*, which differentiate elements with the same name, and *Document Type Definition (DTD)* files and *schema* files, which provide a structural definition for an XML document by specifying the type, order, number and attributes of the elements in an XML document. By defining an XML document's structure, a DTD or Schema reduces the validation and error-checking work of the application using the document. This chapter provides an introduction to an extremely popular XML-related technology—called the *Extensible Stylesheet Language (XSL)*—for transforming XML documents into other document formats such as XHTML. This chapter provides an overview of XML; Chapter 16 discusses XML processing in Python.

Chapter 16—XML Processing

In this chapter, we discuss how Python XML processing and manipulation can be accomplished simply and powerfully using standard and third-party modules. This chapter overviews several ways to process XML documents. The W3C *Document Object Model (DOM)*—an *Application Programming Interface (API)* for XML that is platform and language neutral—is discussed. The DOM API provides a standard set of interfaces (i.e., methods, objects, etc.) for manipulating an XML document's contents. XML documents are hierarchically structured, thus, the DOM represents XML documents as tree structures. Using DOM, programs can modify the content, structure and formatting of documents dynamically. We also present an alternative to DOM called the *Simple API for XML (SAX)*. Unlike DOM, which builds a tree structure in memory, SAX calls specific methods when start tags, end tags, attributes, etc., are encountered in a document. For this reason, SAX is often referred to as an *event-based API*. Python XML support is available through modules **xml.dom.ext** (DOM) and **xml.sax** (SAX). In the chapter, we use *4Suite* (developed by FourThought, Inc.) and PyXML—two collections of Python XML modules. The major feature of this chapter is a case study that uses XML to implement a Web-based message forum.

Chapter 17—Database Application Programming Interface (DB-API)

This chapter enables programs to query and manipulate databases. Most substantial business and Web applications are based on *database management systems (DBMS)*. To support DBMS applications, Python offers the *database application programming interface (DB-API)*. This chapter uses *Structured Query Language (SQL)* to query and manipulate *Relational Database Management Systems (RDBMS)*, specifically a MySQL database. To interface with a MySQL database, Python uses module **MySQLdb**. This chapter contains three examples. The first is a CGI program that displays information about authors, based on criteria provided by the user. The second creates a GUI program that allows the user to enter an SQL query, then displays the results of the query. The third example is a more substantial GUI program that enables the user to maintain a list of contacts. The user can add,

remove, update and find contacts in the database. The exercises ask the reader to modify these programs to provide more functionality, such as verifying that the database does not contain identical entries.

Chapter 18—Process Management

In this chapter, we discuss *concurrency*. Most programming languages provide a simple set of control structures that enable programmers to perform one action at a time and proceed to the next action after the previous one is finished. Such control structures do not allow most programming languages to perform concurrent actions. The kind of concurrency that computers perform today normally is implemented as operating-system *primitives* available only to highly experienced *systems programmers*. Python makes concurrency primitives available to application programmers. We show how to use the **`fork`** command, which creates a new process, and the **`exec`** and **`system`** commands, which execute separate programs. Techniques for controlling input and output with the **`popen`** command are demonstrated and explained. Some of these commands are available on the Unix platform only, so we point this out when appropriate. We also explore Python's cross-platform capabilities through examples that perform specific tasks based on the operating system on which the program is executing. We discuss methods for communicating between processes, including pipes and signals. The signal-handling examples demonstrate how to discover when a user tries to interrupt a program and how to specify an action that the program takes when such an event occurs.

Chapter 19—Multithreading

This chapter introduces *threads*, which are "light-weight processes." They often are more efficient than full-fledged processes created as a result of commands like **`fork`** presented in the previous chapter. We examine basic threading concepts, including the various states in which a thread can exist throughout its life. We discuss how to include threads in a program by subclassing **`threading.Thread`** and overriding method **`run`**. The latter half of the chapter contains examples that address the classic producer/consumer relationship. We develop several solutions to this problem and introduce the concept of thread synchronization and resource allocation. We introduce threading control primitives, such as locks, condition variables, semaphores and events. The final solution uses module **`Queue`** to protect access to shared data stored in a queue. The examples demonstrate the hazards of threaded programs and show how to avoid these hazards. Our solution also demonstrates the value of writing classes for reuse. We reuse our producer and consumer classes to access various synchronized and unsynchronized data types. After completing this chapter, the reader will have many of the tools necessary to write substantial, extensible and professional programs in Python.

Chapter 20—Networking

In this chapter, we explore applications that can communicate over computer networks. A major benefit of a high-level language like Python is that potentially complex topics can be presented and discussed easily through small, working examples. We discuss basic networking concepts and present two examples—a CGI program that displays a chosen Web page in a browser and a GUI example that displays page content (e.g., XHTML) in a text area. We also discuss client-server communication over sockets. The programs in this section demonstrate how to send and receive messages over the network, using connectionless and connection-based protocols. A key feature of the chapter is the live-code implementa-

tion of a collaborative client/server Tic-Tac-Toe game in which two clients play Tic-Tac-Toe by interacting with a multithreaded server that maintains the state of the game. As part of the exercises, readers will write programs that send and receive messages and files. We ask the reader to modify the Tic-Tac-Toe game to determine when a player wins the game.

Chapter 21—Security

This chapter discusses Web programming security issues. Web programming allows the rapid creation of powerful applications, but it also exposes computers to outside attack. We focus on defensive programming techniques that help the programmer prevent security problems by using certain techniques and tools. One of those tools is encryption. We provide an example of encryption and decryption with module **`rotor`**, which acts as a substitution cipher. Another tool is module **`sha`**, which is used to hash values. A third tool is Python's restricted-access (**`rexec`**) module, which creates a restricted environment in which untrusted code can execute without damaging the local computer. This chapter examines technologies, such as *Public Key Cryptography*, *Secure Socket Layer (SSL)*, *digital signatures*, *digital certificates*, *digital steganography* and *biometrics*, which provide network security. Other types of network security, such as firewalls and antivirus programs, are also covered, and common security threats including cryptanalytic attacks, viruses, worms and Trojan horses are discussed.

Chapter 22—Data Structures

Chapter 22 explores the techniques used to create and manipulate standard data structures in Python. Although high-level data types are built into Python, we believe the reader will benefit from this conceptual and programmatic examination of common data structures. The chapter begins with a discussion of self-referential structures and proceeds with a discussion of how to create and maintain various data structures, including *linked lists*, *queues* (or *waiting lines*), *stacks* and *binary trees*. We reuse the linked-list class to implement queues and stacks, so that the code for the inherited class is minimized and emphasis is placed on code reuse. The binary tree class contains methods for pre-, in- and post-order traversals. For each type of data structure, we present complete, working programs and show sample outputs.

Chapter 23—Case Study: Multi-Tier Online Bookstore

This chapter implements an online bookstore that uses MySQL, XML and XSLT to send Web pages to different clients. We begin the chapter with an introduction to an HTTP-session framework that maintains client information over several pages. The client information is "pickled" (serialized) on the server's computer, to be used by the server at a later time. We then discuss WML, a markup language used by wireless clients to pass documents over the Web. Although we demonstrate the application with XHTML, XHTML Basic and WML clients, we designed the bookstore to be extensible, so new client types can be added easily. The Python CGI programs do not change, but the programmer can modify the bookstore to service new clients by simply creating new XML and XSLT documents for those clients. The bookstore program determines the client type and sends the appropriate data to the client. This chapter encompasses many topics from the previous chapters in the book and illustrates a major strength of Python—its ability to integrate several technologies quickly and easily. The topics covered include file processing, serialization (module **`cPickle`**), CGI form processing (module **`cgi`**), database access (module **`MySQLdb`**), XML DOM manipulation and XSLT processing (the 4Suite set of modules.)

Chapter 24—Multimedia
This chapter presents Python's capabilities for making computer applications come alive. It is remarkable that students in entry-level programming courses will be writing Python applications with all these capabilities. Some exciting multimedia applications include *PyOpenGL*, a module that binds Python to OpenGL API to create colorful, interactive graphics; *Alice*, an environment for creating and manipulating 3D graphical worlds in an object-oriented manner; and **`Pygame`**, a large collection of Python modules for creating cross-platform, multimedia applications, such as interactive games. In our PyOpenGL examples, we create rotating objects and three-dimensional shapes. In the Alice example, we create a graphical game version of a popular riddle. The world we create contains a fox, a chicken and a plant. The goal is to move all three objects across a river, without leaving a predator-prey pair alone at any one time. Our first **`Pygame`** example combines **`Tkinter`** and **`Pygame`** to create a GUI compact disc player. The second example illustrates how to play an MPEG movie. The final **`Pygame`** example creates a video game where the user steers a spaceship through an asteroid field to gather energy cells. We discuss many graphics program pitfalls and techniques in the context of this example. With many other programming languages, these projects would be too complex or detailed to present in a book such as this. However, Python's high-level nature, simple syntax and ample modules enable us to present these exciting examples all in the same chapter!

Chapter 25—Python Server Pages (PSP)
In this chapter, we create dynamic Web content using familiar *Extensible HyperText Markup Language (XHTML)* syntax and Python scripts. We discuss both sides of a client-server relationship. The tools used in this chapter include Apache and *Webware for Python*—a suite of software for writing dynamic Web content. An explanation of Python servlets is presented at the beginning of this chapter. In addition to illustrating how PSP handles Python's unique indentation style, our examples illustrate *scriptlets*, *actions* and *directives*. The exercises ask the reader to modify these examples by adding database connections to PSP.

Appendix A—Operator Precedence Chart
This appendix contains the Python operator precedence chart.

Appendix B—ASCII Character Set
Appendix B contains a table of the 128 ASCII alphanumeric symbols.

Appendix C—Number Systems
Appendix C explains the binary, octal, decimal and hexadecimal number systems. We also cover how to convert between bases and perform arithmetic operations in each base.

Appendix D—Python Development Environments
This appendix presents a brief overview of several Python Development environments, including *IDLE*.

Appendix E—Career Resources
This appendix provides resources related to careers in Python and related technologies. The Internet presents valuable resources and services for job seekers and employers. Automatic search features allow employees to scan the Web for open positions. Employers also can find job candidates using the Internet. This reduces the amount of time spent preparing and re-

viewing resumes, and can minimize travel expenses for distance recruiting and interviewing. In this chapter, we explore career services on the Web from the perspectives of job seekers and employers. We introduce comprehensive job sites, industry-specific sites (including sites geared specifically for Python programmers) and contracting opportunities, as well as additional resources and career services designed to meet the needs of a variety of individuals.

Appendix F—Unicode®

This appendix introduces the *Unicode Standard*, an encoding scheme that assigns unique numeric values to the characters of most of the world's languages. It includes a Python program that uses Unicode encoding to print a welcome message in 10 different languages.

Appendices G and H—Introduction to HyperText Markup Language 4: 1 & 2 (on CD)

These appendices provide an introduction to HTML—the HyperText Markup Language. HTML is a markup language for describing the elements of an HTML document (Web page) so that a browser, such as Microsoft's Internet Explorer, can render (i.e., display) that page. These appendices are included for our readers who do not know HTML. Some key topics covered in Appendix G include incorporating text and images in an HTML document, linking to other HTML documents on the Web, incorporating special characters (such as copyright and trademark symbols) into an HTML document and separating parts of an HTML document with horizontal rules. In Appendix H, we discuss more substantial HTML elements and features. We demonstrate how to present information in lists and tables. We discuss how to collect information from people browsing a site. We explain how to use internal linking and image maps to make Web pages easier to navigate. We also discuss how to use frames to display multiple documents in the browser window.

Appendices I and J—Introduction to XHTML: Part 1 & 2

In these appendices, we introduce the *Extensible HyperText Markup Language (XHTML)*. XHTML is a W3C technology designed to replace HTML as the primary means of describing Web content. As an XML-based language, XHTML is more robust and extensible than HTML. XHTML incorporates most of HTML 4's elements and attributes—the focus of these appendices. Appendix I introduces the X*HTML* and write many simple Web pages. We introduce basic XHTML *tags* and *attributes*. A key issue when using XHTML is the separation of the *presentation of a document* (i.e., how the document is rendered on the screen by a browser) from the *structure of that document*. Appendix J continues our XHTML discussion with more substantial XHTML elements and features. We demonstrate how to present information in *lists* and *tables* and discuss how to collect information from people browsing a site. We explain *internal linking* and *image maps*—techniques that make Web pages easier to navigate. We show how to use *frames* to make attractive Web sites.

Appendix K—Cascading Style Sheets™ (CSS)

Appendix K discusses how document authors can control how the browser renders a Web page. In earlier versions of XHTML, Web browsers controlled the appearance (i.e., the rendering) of every Web page. For example, if a document author placed an **`h1`** (i.e., a large heading) element in a document, the browser rendered the element in its own manner, which was often different than the way other Web browsers would render the same document. *Cascading Style Sheets (CSS)* technology allows document authors to specify the styles of their page elements (spacing, margins, etc.) separately from the structure of their documents (sec-

tion headers, body text, links, etc.). This separation of structure from content allows greater manageability and makes changing the style of the document easier and faster.

Appendix L—Accessibility

This appendix discusses how to design accessible Web sites. Currently, the World Wide Web presents challenges to people with various disabilities. Multimedia-rich Web sites hinder text readers and other programs designed to help people with visual impairments, and the increasing amount of audio on the Web is inaccessible to people with hearing impairments. To rectify this situation, the federal government has issued several key legislation that address Web accessibility. For example, the *Americans with Disabilities Act (ADA)* prohibits discrimination on the basis of a disability. The W3C started the *Web Accessibility Initiative (WAI)*, which provides guidelines describing how to make Web sites accessible to people with various impairments. This chapter provides a description of these methods, such as use of the **<headers>** tag to make tables more accessible to page readers, use of the **alt** attribute of the **<img>** tag to describe images, and the proper use of XHTML and related technologies to ensure that a page can be viewed on any type of display or reader. *VoiceXML* also can increase accessibility with speech synthesis and recognition.

Appendix M—HTML/XHTML Special Characters (on CD)

This appendix provides many commonly used HTML/XHTML special characters, called *character entity references*.

Appendix N—HTML/XHTML Colors (on CD)

This appendix lists commonly used HTML/XHTML color names and their corresponding hexadecimal values.

Appendix O—Additional Python 2.2 Features

This book was published as the release of Python 2.2 was impending. We integrated many Python 2.2 features throughout the book. However, there were a few features that we were unable to insert in the text. We assembled these additional features into Appendix O. As you read each chapter, peak ahead to Appendix O for additional discussions and live-code examples.

Resources on Our Web Site

Our Web site, **www.deitel.com**, provides a number of Python-related resources to help you install and configure Python on your Windows or UNIX/Linux systems. The resources include *Installing Python*, *Installing the Apache Web Server*, *Installing MySQL*, *Installing Database Application Programming Interface (DB-API) modules*, *Installing Webware for Python* and *Installing Third-Party Modules*.

Well, there you have it! We have worked hard to create this book and its optional interactive multimedia *Cyber Classroom*. The book is loaded with hundreds of working, Live-Code™ examples, programming tips, self-review exercises and answers, challenging exercises and projects and numerous study aids to help you master the material. The technologies we introduce will help you write Web-based applications quickly and effectively. As you read the book, if something is not clear, or if you find an error, please write to us at **deitel@deitel.com**. We will respond promptly, and we will post corrections and clarifications at **www.deitel.com**.

Prentice Hall maintains **`www.prenhall.com/deitel`**—a Web site dedicated to our Prentice Hall textbooks, multimedia packages and Web-based training products. The site contains "Companion Web Sites" for each of our books that include frequently asked questions (FAQs), downloads, errata, updates, self-test questions and other resources.

Deitel & Associates, Inc., contributes a weekly column to the popular *InformIT* newsletter, currently subscribed to by more than 800,000 IT professionals worldwide. For opt-in registration, visit **`www.InformIT.com`**.

Deitel & Associates, Inc. also offers a free, opt-in newsletter that includes commentary on industry trends and developments, links to articles and resources from published books and upcoming publications, information on future publications, product-release schedules and more. For opt-in registration, visit **`www.deitel.com`**.

You are about to start on a challenging and rewarding path. We hope you enjoy learning with *Python How to Program* as much as we enjoyed writing it!

1.18 Internet and World Wide Web Resources

`www.python.org`

This site is the first place to look for information about Python. The Python home page provides up-to-date news, a FAQ, and a collection of links to Python resources on the Internet including Python software, tutorials, user groups and demos.

`www.zope.com`

`www.zope.org`

Zope is an extensible, open-source Web application server written in Python. It was created by Digital Creations—the company where the Python development team resides.

`www.activestate.com`

ActiveState creates open-source tools for programmers. The company provides a Python distribution called ActivePython and Komodo, an open-source *Integrated Development Environment (IDE)* for many languages, including Python, XML, Tcl and PHP. ActiveState supplies Python tools for Windows and a collection of Python programs called the *Python Cookbook*.

`homepage.ntlworld.com/tibsnjoan/python.html`

This page contains many links to people and groups that develop and use Python.

`www.ddj.com/topics/pythonurl/`

Dr. Dobb's Journal, a programming publication, maintains a list of Python links at this site.

SUMMARY

[Note: Because this Section 1.17 is primarily a summary of the rest of the book, we do not provide summary bullets for that section.]

- Software controls computers (often referred to as hardware).
- A computer is a device capable of performing computations and making logical decisions at speeds millions, even billions, of times faster than human beings can.
- Computers process data under the control of sets of instructions called computer programs. These computer programs guide the computer through orderly sets of actions specified by people called computer programmers.
- The various devices that comprise a computer system (such as the keyboard, screen, disks, memory and processing units) are referred to as hardware.
- The computer programs that run on a computer are referred to as software.

- The input unit is the "receiving" section of the computer. It obtains information (data and computer programs) from various input devices and places this information at the disposal of the other units so that the information may be processed.
- The output unit is the "shipping" section of the computer. It takes information processed by the computer and places it on output devices to make it available for use outside the computer.
- The memory unit is the rapid access, relatively low-capacity "warehouse" section of the computer. It retains information that has been entered through the input unit so that the information may be made immediately available for processing when it is needed and retains information that has already been processed until that information can be placed on output devices by the output unit.
- The arithmetic and logic unit (ALU) is the "manufacturing" section of the computer. It is responsible for performing calculations such as addition, subtraction, multiplication and division and for making decisions.
- The central processing unit (CPU) is the "administrative" section of the computer. It is the computer's coordinator and is responsible for supervising the operation of the other sections.
- The secondary storage unit is the long-term, high-capacity "warehousing" section of the computer. Programs or data not being used by the other units are normally placed on secondary storage devices (such as disks) until they are needed, possibly hours, days, months or even years later.
- Early computers were capable of performing only one job or task at a time. This form of computer operation often is called single-user batch processing.
- Software systems called operating systems were developed to help make it more convenient to use computers. Early operating systems managed the smooth transition between jobs and minimized the time it took for computer operators to switch between jobs.
- Multiprogramming involves the "simultaneous" operation of many jobs on the computer—the computer shares its resources among the jobs competing for its attention.
- Timesharing is a special case of multiprogramming in which dozens or even hundreds of users share a computer through terminals. The computer runs a small portion of one user's job, then moves on to service the next user. The computer does this so quickly that it might provide service to each user several times per second, so programs appear to run simultaneously.
- An advantage of timesharing is that the user receives almost immediate responses to requests rather than having to wait long periods for results, as with previous modes of computing.
- In 1977, Apple Computer popularized the phenomenon of personal computing.
- In 1981, IBM introduced the IBM Personal Computer, legitimizing personal computing in business, industry and government organizations.
- Although early personal computers were not powerful enough to timeshare several users, these machines could be linked together in computer networks, sometimes over telephone lines and sometimes in local area networks (LANs) within an organization. This led to the phenomenon of distributed computing, in which an organization's computing is distributed over networks to the sites at which the real work of the organization is performed.
- Today, information is shared easily across computer networks, where some computers called file servers offer a common store of programs and data that may be used by client computers distributed throughout the network—hence the term client/server computing.
- Computer languages may be divided into three general types: machine languages, assembly languages and high-level languages.
- Any computer can directly understand only its own machine language. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent.

- English-like abbreviations formed the basis of assembly languages. Translator programs called assemblers convert assembly-language programs to machine language at computer speeds.
- Compilers translate high-level language programs into machine-language programs. High-level languages (like Python) contain English words and conventional mathematical notations.
- Interpreter programs directly execute high-level language programs without the need for first compiling those programs into machine language.
- Although compiled programs execute much faster than interpreted programs, interpreters are popular in program-development environments in which programs are recompiled frequently as new features are added and errors are corrected. Interpreters are also popular for developing Web-based applications.
- Objects are essentially reusable software components that model items in the real world. Modular, object-oriented design and implementation approaches make software-development groups more productive than is possible with previous popular programming techniques. Object-oriented programs are often easier to understand, correct and modify than programs developed with earlier methodologies.
- FORTRAN (FORmula TRANslator) was developed by IBM Corporation between 1954 and 1957 for scientific and engineering applications that require complex mathematical computations.
- COBOL (COmmon Business Oriented Language) was developed in 1959 by a group of computer manufacturers and government and industrial computer users. COBOL is used primarily for commercial applications that require precise and efficient manipulation of large amounts of data.
- C evolved from two previous languages, BCPL and B, as a language for writing operating-systems software and compilers.
- Both BCPL and B were “typeless” languages—every data item occupied one “word” in memory and the burden of typing variables fell on the shoulders of the programmer. The C language was evolved from B by Dennis Ritchie at Bell Laboratories.
- Pascal was designed at about the same time as C. It was created by Professor Nicklaus Wirth and was intended for academic use.
- Structured programming is a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify.
- The Ada language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. One important capability of Ada is called multitasking; this allows programmers to specify that many activities are to occur in parallel.
- Most high-level languages generally allow the programmer to write programs that perform only one activity at a time. Python, through techniques called process management and multithreading, enables programmers to write programs with parallel activities.
- Objects are essentially reusable software components that model items in the real world.
- Object technology dates back at least to the mid-1960s. The C++ programming language, developed at AT&T by Bjarne Stroustrup in the early 1980s, is based C and Simula 67.
- In the early 1990s, researchers at Sun Microsystems® developed a purely object-oriented language called Java.
- In the late 1960’s, the Advanced Research Projects Agency of the Department of Defense (ARPA) rolled out the blueprints for networking the main computer systems of about a dozen ARPA-funded universities and research institutions. ARPA proceeded to implement what quickly became called the ARPAnet, the grandparent of today’s Internet.
- Originally designed to connect the main computer systems of about a dozen universities and research organizations, the Internet today is accessible by hundreds of millions of computers worldwide.

- One of ARPA's primary goals for the network was to allow multiple users to send and receive information at the same time over the same communications paths (such as phone lines). The network operated with a technique called packet switching (still in wide use today), in which digital data are sent in small packages called packets. The packets contain data, address information, error-control information and sequencing information. The address information routes the packets of data to their destination. The sequencing information helps reassemble the packets (which—because of complex routing mechanisms—can actually arrive out of order) into their original order for presentation to the recipients.
- The protocol for communicating over the ARPAnet became known as TCP—Transmission Control Protocol. TCP ensured that messages were routed properly from sender to receiver and that those messages arrived intact.
- Bandwidth is the information-carrying capacity of communications lines.
- In 1990, Tim Berners-Lee of CERN (the European Laboratory for Particle Physics) developed the World Wide Web and several communication protocols that form its backbone.
- The Web allows computer users to locate and view multimedia-intensive documents over the Internet.
- Browsers view HTML (Hypertext Markup Language) documents on the World Wide Web.
- Python is a modular extensible language; Python can incorporate new modules (reusable pieces of software).
- The primary distribution center for Python source code, modules and documentation is the Python Web site—**`www.python.org`**—with plans to develop a site dedicated solely to maintaining Python modules.
- Python is portable, practical and extensible.

TERMINOLOGY

Ada
ALU
arithmetic and logic unit (ALU)
assembler
assembly language
batch processing
C
C++
central processing unit (CPU)
clarity
client
client/server computing
COBOL
computer
computer program
computer programmer
data
distributed computing
file server
FORTRAN
function
functionalization
hardware
hardware platform
high-level language
input unit
input/output (I/O)
interpreter
Java
machine dependent
machine independent
machine language
memory
memory unit
multiprocessor
multiprogramming
multitasking
object-oriented programming
output unit
Pascal
Python
personal computer
portability
primary memory
programming language
run a program

screen
software
software reusability
stored program
structured programming
supercomputer
task
terminal
timesharing
top-down, stepwise refinement
translator program
UNIX
workstation

SELF-REVIEW EXERCISES

1.1 Fill in the blanks in each of the following statements:

a) The company that popularized the phenomenon of personal computing was __________.
b) The computer that made personal computing legitimate in business and industry was the __________.
c) Computers process data under the control of sets of instructions called computer __________.
d) The six key logical units of the computer are the __________, __________, __________, __________, __________ and the __________.
e) Python can incorporate new __________ (reusable pieces of software), which can be written by any Python developer.
f) The three classes of languages discussed in the chapter are __________, __________ and __________.
g) The programs that translate high-level language programs into machine language are called __________.
h) C is widely known as the development language of the __________ operating system.
i) In 2001, the core Python development team moved to Digital Creations, the creators of __________—a Web application server written in Python.
j) The Department of Defense developed the Ada language with a capability called __________, which allows programmers to specify activities that can proceed in parallel.

1.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Hardware refers to the instructions that command computers to perform actions and make decisions.
b) The `re` regular-expression module provides pattern-based text manipulation in Python.
c) The ALU provides temporary storage for data that has been entered through the input unit.
d) Software systems called batches manage the transition between jobs.
e) Assemblers convert high-level language programs to assembly language at computer speeds.
f) Interpreter programs compile high-level language programs into machine language faster than compilers.
g) Structured programming is a disciplined approach to writing programs that are clear and easy to modify.
h) Unlike other programming languages, Python is non-extensible.
i) Objects are reusable software components that model items in the real world.
j) Several `Canvas` components include `Label`, `Button`, `Entry`, `Checkbutton` and `Radiobutton`.

ANSWERS TO SELF-REVIEW EXERCISES

1.1 a) Apple. b) IBM Personal Computer. c) programs. d) input unit, output unit, memory unit, arithmetic and logic unit (ALU), central processing unit (CPU), secondary storage unit. e) modules.

f) machine languages, assembly languages, high-level languages. g) compilers. h) UNIX. i) Zope. j) multitasking.

1.2 a) False. Software refers to the instructions that control computers, also referred to as hardware. Hardware refers to the computer's devices. b) True. c) False. The memory unit provides temporary storage for data that have been entered through the input unit. The arithmetic and logic unit (ALU) performs the calculations and contains the decision mechanisms of the computer. d) False. Software systems called operating systems manage the transition between jobs; in single-user batch processing, the computer runs a single program at a time while processing data in batches. e) False. Assemblers convert assembly-language programs to machine language at computer speeds. f) False. Interpreter programs can directly execute high-level language programs without compiling them into machine language. g) True. h) False. Unlike other programming languages, Python is extensible. i) True. j) False. Several **`Tkinter`** components include **`Label`**, **`Button`**, **`Entry`**, **`Checkbutton`** and **`Radiobutton`**.

EXERCISES

1.3 Categorize each of the following items as either hardware or software:
a) CPU.
b) ALU.
c) Input unit.
d) A word-processor program.
e) Python modules.

1.4 Translator programs, such as assemblers and compilers, convert programs from one language (referred to as the *source* language) to another language (referred to as the *object* language). Determine which of the following statements are *true* and which are *false*:
a) A compiler translates high-level language programs into object language.
b) An assembler translates source-language programs into machine-language programs.
c) A compiler converts source-language programs into object-language programs.
d) High-level languages are generally machine dependent.
e) A machine-language program requires translation before it can be run on a computer.

1.5 Fill in the blanks in each of the following statements:
a) Python can provide information about itself, a technique called __________.
b) A computer program that converts assembly-language programs to machine language programs is called __________.
c) The logical unit of the computer that receives information from outside the computer for use by the computer is called __________.
d) The process of instructing the computer to solve specific problems is called __________.
e) Three high-level Python data types are: __________, __________ and __________.
f) __________ is the logical unit of the computer that sends information that has already been processed by the computer to various devices so that the information may be used outside the computer.
g) The general name for a program that converts programs written in a certain computer language into machine language is __________.

1.6 Fill in the blanks in each of the following statements:
a) __________ is the logical unit of the computer that retains information.
b) __________ is the logical unit of the computer that makes logical decisions.
c) The commonly used abbreviation for the computer's control unit is __________.
d) The level of computer language most convenient to the programmer for writing programs quickly and easily is __________.
e) __________ are "mappable" types—keys are stored with their associated values.

f) The only language that a computer can understand directly is called that computer's __________.
g) The __________ is the logical unit of the computer that coordinates the activities of all the other logical units.

1.7 What do each of the following acronyms stand for:
a) W3C.
b) XML.
c) DB-API.
d) CGI.
e) XHTML.
f) TCP/IP.
g) PSP.
h) Tcl/Tk.
i) SSL.
j) HMD.

1.8 State whether each of the following is *true* or *false*. If *false*, explain your answer.
a) Inheritance is a form of software reusability in which new classes are developed quickly and easily by absorbing the capabilities of existing classes and adding appropriate new capabilities.
b) **`Pmw`** is a module that provides an interface to the popular Tcl/Tk graphical-user-interface toolkit.
c) Like other high-level languages, Python is generally considered to be machine-independent.

2

Introduction to Python Programming

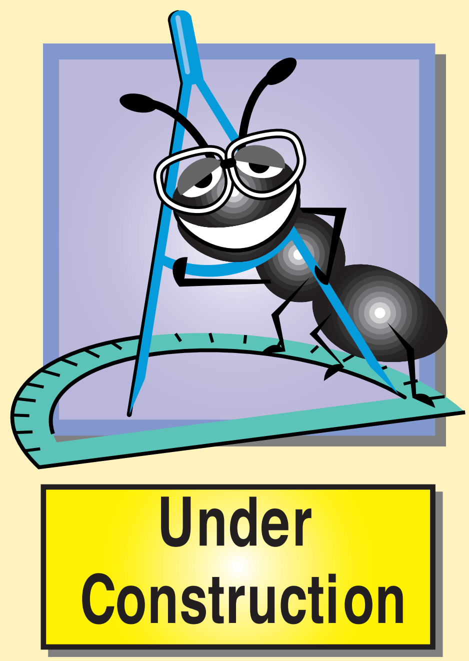
Objectives

- To understand a typical Python program-development environment.
- To write simple computer programs in Python.
- To use simple input and output statements.
- To become familiar with fundamental data types.
- To use arithmetic operators.
- To understand the precedence of arithmetic operators.
- To write simple decision-making statements.

High thoughts must have high language.
Aristophanes

Our life is frittered away by detail…Simplify, simplify.
Henry Thoreau

My object all sublime
I shall achieve in time.
W.S. Gilbert



Outline

2.1 Introduction

Python facilitates a disciplined approach to computer-program design. In this first programming chapter, we introduce Python programming and present several examples that illustrate important features of the language. To understand each example, we analyze the code one statement at a time. After presenting basic concepts in this chapter, we examine the *structured programming* approach in Chapters 3–5. At the same time that we explore introductory Python topics, we also begin our discussion of object-oriented programming—the key programming methodology presented throughout this text. For this reason, we conclude this chapter with Section 2.10, Thinking About Objects.

2.2 First Program in Python: Printing a Line of Text[1]

We begin by considering a simple program that prints a line of text. Figure 2.1 illustrates the program and its screen output.

```
# Fig. 2.1: fig02_01.py
# Printing a line of text in Python.

print "Welcome to Python!"
```

```
Welcome to Python!
```

Fig. 2.1 Text-printing program.

1. The resources for this book, including step-by-step instructions for installing Python on Windows and Unix/Linux platforms, are posted at **www.deitel.com**.

This program illustrates several important features of the Python language. Let us consider each line of the program. Each program we present in this book has line numbers included for the reader's convenience; line numbers are not part of actual Python programs. Line 4 does the "real work" of the program, namely displaying the phrase **`Welcome to Python!`** on the screen. However, let us consider each line in order.

Lines 1–2 begin with the pound symbol (**`#`**), which indicates that the remainder of each line is a *comment*. Programmers insert comments to *document* programs and to improve program readability. Comments also help other programmers read and understand your program. Comments do not cause the computer to perform any action when the program is run—Python ignores comments. We begin every program with a comment indicating the figure number and the file name in which that program is stored (line 1). We can place any text we choose in comments. All of the Python programs for this book are included on the enclosed CD and also are available free for download at **`www.deitel.com`**.

A comment that begins with **`#`** is called a *single-line comment*, because the comment terminates at the end of the current line. A **`#`** comment also can begin in the middle of a line and continue until the end of that line. Such a comment typically documents the Python code that appears at the beginning of that line. Unlike other programming languages, Python does not have a separate symbol for a multiple-line comment, so each line of multiple-line comment must start with the **`#`** symbol. The comment text "**`Printing a line of text in Python.`**" describes the purpose of the program (line 2).

Good Programming Practice 2.1

Place abundant comments throughout a program. Comments help other programmers understand the program, assist in debugging *a program (i.e., discovering and removing errors in a program) and list useful information. Comments also help you understand your programs when you revisit the code for modifications or updates.*

Good Programming Practice 2.2

Every program should begin with a comment describing the purpose of the program.

Line 3 is simply a blank line. Programmers use blank lines and space characters to make programs easier to read. Together, blank lines, space characters and tab characters are known as *white space*. (Space characters and tabs are known specifically as *white-space characters*.) Blank lines are ignored by Python.

Good Programming Practice 2.3

Use blank lines to enhance program readability.

The Python **`print`** command (line 4) instructs the computer to display the *string* of characters contained between the quotation marks. A string is a sequence of characters contained inside double quotes. The entire line is called a *statement*. In some programming languages, like C++ and Java, statements must end with a semicolon. In Python, most statements simply end when the lines on which they are written end. When the statement on line 4 executes, it displays the message **`Welcome to Python!`** on the screen. Note that the double quotes that delineate the string do not appear in the output.

Output (i.e., displaying information) and input (i.e., receiving information) in Python are accomplished with *streams* of characters. When the preceding statement executes, it

sends the stream of characters **`Welcome to Python!`** to the *standard output stream*. The standard output stream is the channel through which an application presents information to the user—this information typically is displayed on the screen, but may be printed on a printer, written to a file, etc. It may even be spoken or issued to braille devices, so users with visual impairments can receive the outputs.

Python statements can be executed two ways. The first is by typing statements into an editor to create a program and saving the file with a **`.py`** extension (as in Fig. 2.1). Python files typically end with **`.py`**, although other extensions (e.g., **`.pyw`** on Windows) can be used. To use the Python interpreter to *execute* (run) the program in the file, type

```
python file.py
```

at the DOS or Unix *shell command line*, in which *file*.**`py`** is the name of the Python file. The shell command line is a text "terminal" in which the user can type commands that cause the computer system to respond. [*Note:* To invoke Python, the system path variable must be set properly to include the **`python`** *executable*—a file containing the Python interpreter program that can be run. The resources for this book—posted at our Web site **`www.deitel.com`**—include instructions on how to set the appropriate system path variable.]

When the Python interpreter runs a program stored in the file, the interpreter starts at the first line of the file and executes statements until the end of the file. The output box in Fig. 2.1 contains the results of the Python interpreter running **`fig02_01.py`**.

The second way to execute Python statements is *interactively*. Typing

```
python
```

at the shell command line runs the Python interpreter in *interactive mode*. With this mode, the programmer types statements directly to the interpreter, which executes these statements one at a time.

Testing and Debugging Tip 2.1

In interactive mode, Python statements are entered and interpreted one at a time. This mode often is useful when debugging a program.

Testing and Debugging Tip 2.2

When the Python interpreter is invoked on a file, the interpreter exits after the last statement in the file executes. However, invoking the interpreter on a file using the **`-i`** *flag (for example,* **`python -i file.py`***) causes the interpreter to enter interactive mode after executing the statements in the file. This is useful when debugging a program.*

Figure 2.2 shows Python 2.2 running in interactive mode on Windows. The first three lines display information about the version of Python being used (2.2b2 means "version 2.2 beta 2"). The fourth line contains the *Python prompt* (**`>>>`**). When a programmer types a statement at the Python prompt and presses the *Enter* key (sometimes labeled the *Return* key), the interpreter executes the statement.

The **`print`** statement on the fifth line of Fig. 2.2 displays the text **`Welcome to Python!`** to the screen (note, again, that the double quotes delineating the screen do not print). After printing the text to the screen, the interpreter waits for the user to enter the next statement. We exit interactive mode by typing the *Ctrl-Z* end-of-file character (on Microsoft Windows systems) and pressing the *Enter* key. Figure 2.3 lists the keyboard combinations for the end-of-file character for various computer systems.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> print "Welcome to Python!"
Welcome to Python!
>>> ^Z
```

Fig. 2.2 Interactive mode. (Python interpreter software Copyright © 2001 Python Software Foundation.)

2.3 Modifying our First Python Program

This section continues our introduction to Python programming with two examples that modify Fig. 2.1 to display text on one line using multiple statements and to display text on several lines using a single statement.

2.3.1 Displaying a Single Line of Text with Multiple Statements

`Welcome to Python!` can be printed in several ways. For example, Fig. 2.4 uses two `print` statements (lines 4–5), yet produces the same output as the program in Fig. 2.1. Most of the program is identical to that of Fig. 2.1, so we discuss only the changes here.

Line 4 displays the string `"Welcome"`. Normally, after the `print` statement displays its string, Python begins a new line—subsequent outputs are displayed on the line or lines that follow the `print` statement's string. However, the comma (`,`) at the end of line 4 tells Python not to begin a new line but instead to add a space after the string; thus, the next string the program displays (line 5) appears on the same line as the string `"Welcome"`.

Computer system	Keyboard combination
UNIX/Linux systems	*Ctrl-D* (on a line by itself)
DOS/Windows	*Ctrl-Z* (sometimes followed by pressing *Enter*)
Macintosh	*Ctrl-D*

Fig. 2.3 End-of-file key combinations for various popular computer systems.

```
1  # Fig. 2.4: fig02_04.py
2  # Printing a line with multiple statements.
3
4  print "Welcome",
5  print "to Python!"
```

```
Welcome to Python!
```

Fig. 2.4 Printing one line using several `print` statements.

2.3.2 Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines using *newline characters*. Newline characters are “special characters” that position the screen cursor to the beginning of the next line. Figure 2.5 outputs four lines of text, using newline characters to determine when to begin each new line.

Most of the program is identical to those of Fig. 2.1 and Fig. 2.4, so we discuss only the changes here. Line 4 displays four separate lines of text to the screen. Normally, the characters in a string display exactly as they appear in the double quotes. Notice, however, that the two characters **** and **n** (which appear three times in line 4) do not appear in the output. Python offers *special characters* that perform certain tasks, such as backspace and carriage return. A special character is formed by combining the backslash (****) character, also called the *escape character*, with a letter. When a backslash exists in a string of characters, the backslash and the character immediately following the backslash form an *escape sequence*. An example of an escape sequence is **\n**, which represents the newline character. Each occurrence of the **\n** escape sequence causes the screen cursor that controls where the next character will appear to move to the beginning of the next line. To print a blank line, simply place two newline characters back-to-back. Figure 2.6 lists other common escape sequences.

```
1  # Fig. 2.5: fig02_05.py
2  # Printing multiple lines with a single statement.
3
4  print "Welcome\nto\n\nPython!"
```

```
Welcome
to

Python!
```

Fig. 2.5 Printing multiple lines using a single **print** statement.

Escape Sequence	Description
`\n`	Newline. Move the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Move the screen cursor to the beginning of the current line; do not advance to the next line.
`\b`	Backspace. Move the screen cursor back one space.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Print a backslash character.
`\"`	Double quote. Print a double quote character.
`\'`	Single quote. Print a single quote character.

Fig. 2.6 Escape sequences.

2.4 Another Python Program: Adding Integers

Our next program inputs two integers (whole numbers, like –22, 7 and 1024) typed by a user at the keyboard, computes the sum of the values and displays the result. This program invokes Python functions ***`raw_input`*** and ***`int`*** to obtain the two integers. Again, the program uses the **`print`** statement to display the sum of the integers. Figure 2.7 contains the program and its output.

Lines 1–2 contain comments that state the figure number, file name and the purpose of the program. Line 5 calls Python's *built-in function* **`raw_input`** to request user input. A built-in function is a piece of code provided by Python that performs a task. The task is performed by *calling the function*—writing the function name, followed by parentheses (**`()`**). After performing its task, a function may return a value that represents the result of the task. We study functions in depth in Chapter 4, where we mention many other built-in functions and show how programmers can create their own *programmer-defined functions*.

Python function **`raw_input`** takes the *argument*, **`"Enter first integer:\n"`** that requests user input. An argument is a value that a function accepts and uses to perform its task. In this case, function **`raw_input`** accepts the "prompt" argument (that requests user input) and displays that prompt to the screen. In response to viewing this prompt, the user enters a number and presses the *Enter* key—this sends the number to function **`raw_input`** in the form of a string.

The result of **`raw_input`** (a string containing the characters typed by the user) is assigned to *variable* **`integer1`** using the *assignment symbol*, **`=`**. In Python, variables are more specifically referred to as *objects*. An object resides in the computer's memory and contains information used by the program. The term object normally implies that *attributes* (data) and *behaviors* (methods) are associated with the object. The object's methods use the attributes to perform tasks. A variable name (e.g., **`integer1`**) consists of letters, digits and underscores (_) and does not begin with a digit. Python is *case sensitive*—uppercase and lowercase letters are different, so **`a1`** and **`A1`** are different variables. An object can have multiple names, called *identifiers*. Each identifier (or variable name) *references* (points to) the object (or variable) in memory. The statement in line 5 is normally read as "Variable **`integer1`** is assigned the value returned by **`raw_input( "Enter first integer:\n" )`**." The actual meaning of such a line, however, is "**`integer1`** references the value returned by **`raw_input( "Enter first integer:\n" )`**."

```
1  # Fig. 2.7: fig02_07.py
2  # Simple addition program.
3
4  # prompt user for input
5  integer1 = raw_input( "Enter first integer:\n" )  # read string
6  integer1 = int( integer1 )   # convert string to integer
7
8  integer2 = raw_input( "Enter second integer:\n" ) # read string
9  integer2 = int( integer2 )   # convert string to integer
10
11 sum = integer1 + integer2    # compute and assign sum
12
13 print "Sum is", sum          # print sum
```

Fig. 2.7 Addition program. (Part 1 of 2.)

```
Enter first integer:
45
Enter second integer:
72
Sum is 117
```

Fig. 2.7 Addition program. (Part 2 of 2.)

Good Programming Practice 2.4

Choosing meaningful variable names helps a program to be "self-documenting," i.e., it is easier to understand the program simply by reading it, rather than having to read manuals or use excessive comments.

Good Programming Practice 2.5

Avoid identifiers that begin with underscores and double underscores, because the Python interpreter or other Python code may reserve those characters for internal use. This prevents names you choose from being confused with names the interpreter chooses.

In addition to a name and value, each object has a *type*. An object's type identifies the kind of information (e.g., integer, string, etc.) stored in the object. Integers are whole numbers that encompass negative numbers (–14), zero (0) and positive numbers (6). In languages like C++ and Java, the programmer must *declare* (state) the object type before using the object in the program. However, Python uses *dynamic typing*, which means that Python determines an object's type during program execution. For example, if object **a** is initialized to **2**, then the object is of type "integer" (because the number 2 is an integer). Similarly, if object **b** is initialized to **"Python"**, then the object is of type "string." Function **raw_input** returns values of type "string," so the object referenced by **integer1** (line 5) is of type "string."

To perform integer addition on the value referenced by **integer1**, the program must convert the string value to an integer value. Python function ***int*** (line 6) converts a string or a number to an integer value and returns the new value. If we do not obtain an integer value for variable **integer1**, we will not achieve the desired results—the program would combine the two strings instead of adding two integers. Figure 2.8 demonstrates this with an interactive session.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> value1 = raw_input( "Enter an integer: " )
Enter an integer: 2
>>> value2 = raw_input( "Enter an integer: " )
Enter an integer: 4
>>> print value1 + value2
24
```

Fig. 2.8 Adding values from **raw_input** (incorrectly) without converting to integers (the result should be 6).

The *assignment statement* (line 11 of Fig. 2.7) calculates the sum of the variables **integer1** and **integer2** and assigns the result to variable **sum**, using the assignment symbol **=**. The statement is read as, "**sum** references the value of **integer1 + integer2**." Most calculations are performed through assignment statements.

The **+** symbol is an *operator*—a special symbol that performs a specific operation. In this case, the **+** operator performs addition. The **+** operator is called a *binary operator*, because it has two *operands* (values) on which it performs its operation. In this example, the operands are **integer1** and **integer2**. [*Note*: In Python, the **=** symbol is not an operator. Rather, it is referred to as the assignment symbol.]

Common Programming Error 2.1

Trying to access a variable that has not been given a value is a run-time error.

Good Programming Practice 2.6

Place spaces on either side of a binary operator or symbol. This helps the operator or symbol stand out, making the program more readable.

Line 13 displays the string **"Sum is"** followed by the numerical value of variable **sum**. Items we want to output are separated by commas (**,**). Note that this **print** statement outputs values of different types, namely a string and an integer.

Calculations also can be performed in output statements. We could have combined the statements in lines 11 and 13 into the statement

```
print "Sum is", integer1 + integer2
```

thus eliminating the need for variable **sum**. You should make such combinations only if you feel it makes your programs clearer.

2.5 Memory Concepts

Variable names such as **integer1**, **integer2** and **sum** actually correspond to Python objects. Every object has a *type*, a *size*, a *value* and a *location* in the computer's memory. A program cannot change an object's type or location. Some object types permit programmers to change the object's value. We discuss these types beginning in Chapter 5, Tuples, Lists and Dictionaries.

When the addition program in Fig. 2.7, executes the statement

```
integer1 = raw_input( "Enter first integer:\n" )
```

Python first creates an object to hold the user-entered string and places the object into a memory location. The **=** assignment symbol then binds (associates) the name **integer1** with the newly created object. Suppose the user enters **45** at the **raw_input** prompt. Python places the string **"45"** into memory at a starting location to which the name **integer1** is bound, as shown in Fig. 2.9. When the statement

```
integer1 = int( integer1 )
```

executes, function **int** creates a new object to store the integer value **45.** This integer object begins at a new memory location and Python binds the name **integer1** to this new memory location (Fig. 2.10). Variable **integer1** no longer refers to the memory location that contains the string value **"45"**.

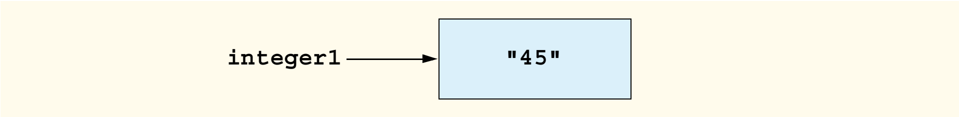


Fig. 2.9 Memory location showing value of a variable and the name bound to the value.

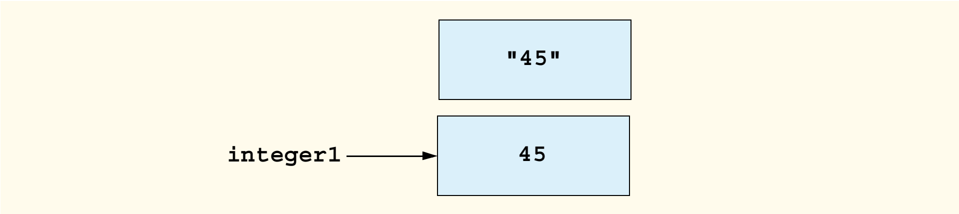


Fig. 2.10 Memory location showing the name and value of a variable.

Returning to our addition program, when the statements

```
integer2 = raw_input( "Enter second integer:\n" )
integer2 = int( integer2 )
```

execute, suppose the user enters the string **"72"**. After the program converts this value to the integer value **72** and places the value into a memory location to which **integer2** is bound, memory appears as in Fig. 2.11. Note that the locations of these objects are not necessarily adjacent in memory.

Once the program has obtained values for **integer1** and **integer2**, the program adds these values and assigns the sum to variable **sum**. After the statement

```
sum = integer1 + integer2
```

performs the addition, memory appears as in Fig. 2.12. Note that the values of **integer1** and **integer2** appear exactly as they did before they were used in the calculation of **sum**. These values were used, but not modified, as the computer performed the calculation. Thus, when a value is read out of a memory location, the value is not changed.

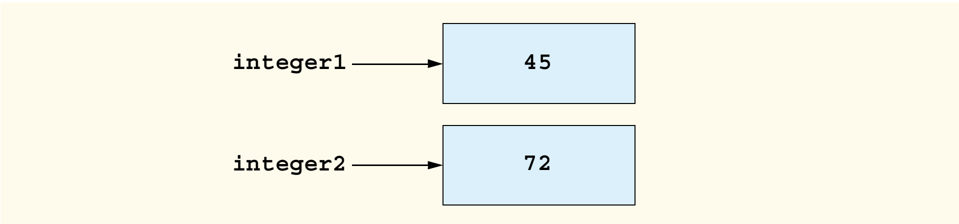


Fig. 2.11 Memory locations after values for two variables have been input.

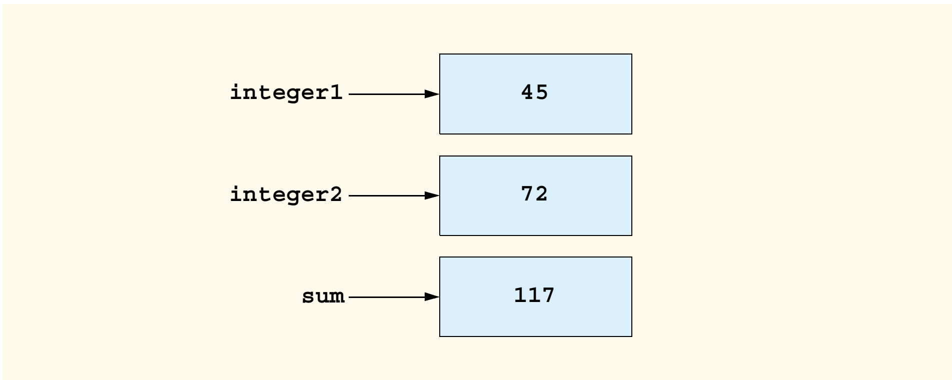


Fig. 2.12 Memory locations after a calculation.

Figure 2.13 demonstrates that each Python object has a location, a type and a value and that these object *properties* are accessed through an object’s name. This program is identical to the program in Fig. 2.7, except that we have added statements that display the memory location, type and value for each object at various points in the program.

```
# Fig. 2.13: fig02_13.py
# Displaying an object’s location, type and value.

# prompt the user for input
integer1 = raw_input( "Enter first integer:\n" )  # read a string
print "integer1: ", id( integer1 ), type( integer1 ), integer1
integer1 = int( integer1 )    # convert the string to an integer
print "integer1: ", id( integer1 ), type( integer1 ), integer1

integer2 = raw_input( "Enter second integer:\n" ) # read a string
print "integer2: ", id( integer2 ), type( integer2 ), integer2
integer2 = int( integer2 )    # convert the string to an integer
print "integer2: ", id( integer2 ), type( integer2 ), integer2

sum = integer1 + integer2     # assignment of sum
print "sum: ", id( sum ), type( sum ), sum
```

```
Enter first integer:
5
integer1:  7956744 <type 'str'> 5
integer1:  7637688 <type 'int'> 5
Enter second integer:
27
integer2:  7776368 <type 'str'> 27
integer2:  7637352 <type 'int'> 27
sum:  7637436 <type 'int'> 32
```

Fig. 2.13 Object’s location, type and value.

Line 6 prints **integer1**'s location, type and value after the call to **raw_input**. Python function ***id*** returns the interpreter's representation of the variable's location. Function ***type*** returns the type of the variable. We print these values again (line 8), after converting the string value in **integer1** to an integer value. Notice that both the type and the location of variable **integer1** change as a result of the statement

```
integer1 = int( integer1 )
```

The change underscores the fact that a program cannot change a variable's type. Instead, the statement causes Python to create a new integer value in a new location and assigns the name **integer1** to this location. The location to which **integer1** previously referred is no longer accessible. The remainder of the program prints the location type and value for variables **integer2** and **sum** in a similar manner.

2.6 Arithmetic

Many programs perform arithmetic calculations. Figure 2.14 summarizes the *arithmetic operators*. Note the use of various special symbols not used in algebra. The *asterisk (*****)* indicates multiplication and the *percent sign (**%**)* is the *modulus operator* that we discuss shortly. The arithmetic operators in Fig. 2.14 are binary operators, (i.e., operators that take two operands). For example, the expression **integer1 + integer2** contains the binary operator **+** and the two operands **integer1** and **integer2**.

Python is an evolving language, and as such, some of its features change over time. Starting with Python 2.2, the behavior of the **/** division operator will begin to change from "floor division" to "true division." *Floor division* (sometimes called *integer division*), divides the numerator by the denominator and returns the highest integer value that is not greater than the result. For example, dividing 7 by 4 with floor division yields 1 and dividing 17 by 5 with floor division yields 3. Note that any fractional part in floor division is simply discarded (i.e., *truncated*)—no rounding occurs. *True division* yields the precise *floating-point* (i.e., numbers with a decimal point such as 7.0, 0.0975 and 100.12345) result of dividing the numerator by the denominator. For example, dividing 7 by 4 with true division yields 1.75.

Python operation	Arithmetic operator	Algebraic expression	Python expression
Addition	**+**	$f + 7$	**f + 7**
Subtraction	**-**	$p - c$	**p - c**
Multiplication	*****	bm	**b * m**
Exponentiation	******	x^y	**x ** y**
Division	**/** **//** (new in Python 2.2)	x / y or $\frac{x}{y}$ or $x \div y$	**x / y** **x // y**
Modulus	**%**	$r \bmod s$	**r % s**

Fig. 2.14 Arithmetic operators.

In prior versions, Python contained only one operator for division—the / operator. The behavior (i.e., floor or true division) of the operator is determined by the type of the operands. If the operands are both integers, the operator performs floor division. If one or both of the operands are floating-point numbers, the operator performs true division.

The language designers and many programmers disliked the ambiguity of the / operator and decided to create two operators for version 2.2—one for each type of division. The / operator performs true division and the // operator performs floor division. However, this decision could introduce errors into programs that use older versions of Python. Therefore, the designers came up with a compromise: Starting with Python 2.2 all future 2.x versions will include two operators, but if a program author wants to use the new behavior, the programmer must state their intention explicitly with the statement

```
from __future__ import division
```

After Python sees this statement, the / operator performs true division and the // operator performs floor division. The interactive session in Fig. 2.15 demonstrates floor division and true division.

We first evaluate the expression `3 / 4`. This expression evaluates to the value `0`, because the default behavior of the / operator with integer operands is floor division. The expression `3.0 / 4.0` evaluates to `0.75`. In this case, we use floating-point operands, so the / operator performs true division. The expressions `3 // 4` and `3.0 // 4.0` evaluate to `0` and `0.0`, respectively, because the // operator always performs floor division, regardless of the types of the operands. Then, in line 13 of the interactive session, we change the behavior of the / operator with the special `import` statement. In effect, this statement turns on the true division behavior for operator /. Now the expression `3 / 4` evaluates to `0.75`. [*Note*: In this text, we use only the default 2.2 behavior for the / operator, namely floor division for integers (lines 5–6 of Fig. 2.15) and true division for floating-point numbers (lines 7–8 of Fig. 2.15).]

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> 3 / 4       # floor division (default behavior)
0
>>> 3.0 / 4.0   # true division (floating-point operands)
0.75
>>> 3 // 4      # floor division (only behavior)
0
>>> 3.0 // 4.0  # floating-point floor division
0.0
>>> from __future__ import division
>>> 3 / 4       # true division (new behavior)
0.75
>>> 3.0 / 4.0   # true division (same as before)
0.75
```

Fig. 2.15 Difference in behavior of the / operator.



Portability Tip 2.1

In Python version 3.0 (due to be released no sooner than 2003), the **/** *operator can perform only true division. After the release of version 3.0, programmers need to update applications to compensate for the new behavior. For more information on this future change, see* ***python.sourceforge.net/peps/pep-0238.html***

Python provides the modulus operator (**%**), which yields the remainder after integer division. The expression **x % y** yields the remainder after **x** is divided by **y**. Thus, **7 % 4** yields **3** and **17 % 5** yields **2**. This operator is most commonly used with integer operands, but also can be used with other arithmetic types. In later chapters, we discuss many interesting applications of the modulus operator, such as determining whether one number is a multiple of another. (A special case of this is determining whether a number is odd or even.) [*Note*: The modulus operator can be used with both integer and floating-point numbers.]

Arithmetic expressions in Python must be entered into the computer in *straight-line form*. Thus, expressions such as "**a** divided by **b**" must be written as **a / b**, so that all constants, variables and operators appear in a straight line. The algebraic notation

$$\frac{a}{b}$$

is generally not acceptable to compilers or interpreters, although some special-purpose software packages do exist that support more natural notation for complex mathematical expressions.

Parentheses are used in Python expressions in much the same manner as in algebraic expressions. For example, to multiply **a** times the quantity **b + c**, we write

```
a * (b + c)
```

Python applies the operators in arithmetic expressions in a precise sequence determined by the following *rules of operator precedence*, which are generally the same as those followed in algebra:

1. Expressions contained within pairs of parentheses are evaluated first. Thus, parentheses may force the order of evaluation to occur in any sequence desired by the programmer. Parentheses are said to be at the "highest level of precedence." In cases of *nested*, or *embedded*, parentheses, the operators in the innermost pair of parentheses are applied first.

2. Exponentiation operations are applied next. If an expression contains several exponentiation operations, operators are applied from right to left.

3. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are applied from left to right. Multiplication, division and modulus are said to be on the same level of precedence.

4. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from left to right. Addition and subtraction also have the same level of precedence.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the expression

```
a * (b + c) + c * (d + e)
```

does not contain nested parentheses. Rather, the parentheses in this expression are said to be “on the same level.”

When we say that certain operators are applied from left to right, we are referring to the *associativity* of the operators. For example, in the expression

```
a + b + c
```

the addition operators (**+**) associate from left to right. We will see that some operators associate from right to left.

Figure 2.16 summarizes these rules of operator precedence. This table will be expanded as additional Python operators are introduced. A complete precedence chart is included in the appendices.

Now let us consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its Python equivalent. The following is an example of an arithmetic mean (average) of five terms:

Algebra: $$m = \frac{a + b + c + d + e}{5}$$

Python: **`m = ( a + b + c + d + e ) / 5`**

The parentheses are required because division has higher precedence than addition and, hence, the division will be applied first. The entire quantity **`( a + b + c + d + e )`** is to be divided by **`5`**. If the parentheses are erroneously omitted, we obtain **`a + b + c + d + e / 5`**, which evaluates incorrectly as

$$a + b + c + d + \frac{e}{5}$$

The following is an example of the equation of a straight line:

Algebra: $y = mx + b$

Python: **`y = m * x + b`**

No parentheses are required. The multiplication is applied first, because multiplication has a higher precedence than addition.

The following example contains modulus (**%**), multiplication, division, addition and subtraction operations:

Algebra: $z = pr\%q + w/x - y$

Python:

```
z  =  p  *  r  %  q  +  w  /  x  -  y
         1     2     4     3     5
```

Operator(s)	Operation(s)	Order of Evaluation (Precedence)
`( )`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`**`	Exponentiation	Evaluated second. If there are several, they are evaluated right to left.
`* / // %`	Multiplication Division Modulus	Evaluated third. If there are several, they are evaluated left to right. [*Note:* The `//` operator is new in version 2.2]
`+ -`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Fig. 2.16 Precedence of arithmetic operators.

The circled numbers under the statement indicate the order in which Python applies the operators. The multiplication, modulus and division are evaluated first, in left-to-right order (i.e., they associate from left to right) because they have higher precedence than addition and subtraction. The addition and subtraction are applied next. These are also applied left to right. Once the expression has been evaluated, Python assigns the result to variable **`z`**.

To develop a better understanding of the rules of operator precedence, consider how a second-degree polynomial is evaluated:

```
y  =  a  *  x  ** 2 +  b  *  x  +  c
         2     1   4     3     5
```

The circled numbers under the statement indicate the order in which Python applies the operators.

Suppose variables **`a`**, **`b`**, **`c`** and **`x`** are initialized as follows: **`a = 2`**, **`b = 3`**, **`c = 7`** and **`x = 5`**. Figure 2.17 illustrates the order in which the operators are applied in the preceding second-degree polynomial.

The preceding assignment statement can be parenthesized with unnecessary parentheses, for clarity, as

```
y = ( a * ( x ** 2 ) ) + ( b * x ) + c
```

Good Programming Practice 2.7

As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. These parentheses are called redundant parentheses. Redundant parentheses are commonly used to group subexpressions in a large expression to make that expression clearer. Breaking a large statement into a sequence of shorter, simpler statements also promotes clarity.

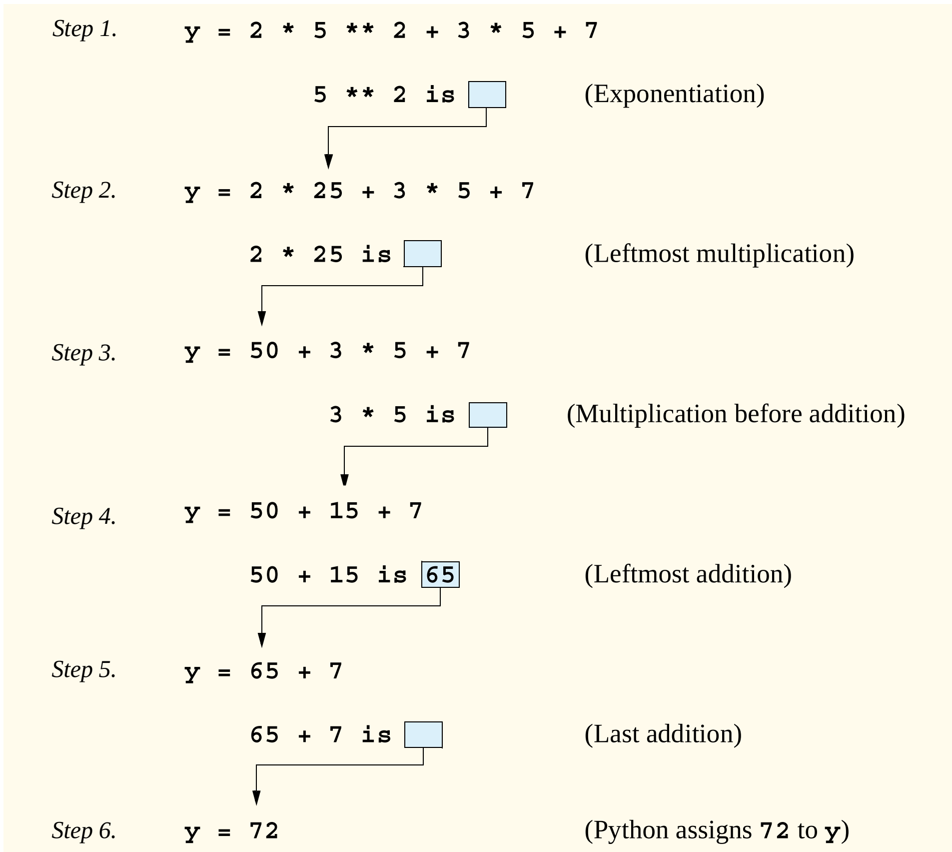


Fig. 2.17 Order in which a second-degree polynomial is evaluated.

2.7 String Formatting

Now that we have investigated numeric values, let us turn our attention to strings. Unlike some other popular programming languages, Python provides strings as a built-in data type, thereby enabling Python programs to perform powerful text-based operations easily. We have already learned how to create a string by placing text inside double quotes (`"`). Python strings can be created in a variety of other ways, as Fig. 2.18 demonstrates.

Line 4 creates a string with the familiar double-quote character (`"`). If we want such a string to print double quotes to the screen, we must use the escape sequence for the double-quote character (`\"`), rather than the double-quote character itself.

Strings also can be created using the single-quote character (`'`) as shown in line 5. If we want to use the double-quote character inside a string created with single quotes, we do not need to use the escape character. Similarly, if we want to use a single-quote character inside a string created with double quotes, we do not need to use the escape sequence (line 7). However, if we want to use the single-quote character inside a string created with single quotes (line 6), we must use the escape sequence (`\'`).

```
1  # Fig. 2.18: fig02_18.py
2  # Creating strings and using quote characters in strings.
3
4  print "This is a string with \"double quotes.\""
5  print 'This is another string with "double quotes."'
6  print 'This is a string with \'single quotes.\''
7  print "This is another string with 'single quotes.'"
8  print """This string has "double quotes" and 'single quotes'.
9     You can even do multiple lines."""
10 print '''This string also has "double" and 'single' quotes.'''
```

```
This is a string with "double quotes."
This is another string with "double quotes."
This is a string with 'single quotes.'
This is another string with 'single quotes.'
This string has "double quotes" and 'single quotes'.
   You can even do multiple lines.
This string also has "double" and 'single' quotes.
```

Fig. 2.18 Creating Python strings.

Python also supports *triple-quoted strings* (lines 8–10). Triple-quoted strings are useful for programs that output strings with special characters, such as quote characters. Single- or double-quote characters inside a triple-quoted string do not need to use the escape sequence. Triple-quoted strings also are used for large blocks of text, because triple-quoted strings can span multiple lines. We use triple-quoted strings in this book when we write programs that output large blocks of text for the Web.

Python strings support simple, but powerful, output formatting. We can create strings that format output in several ways:

1. Rounding floating-point values to an indicated number of decimal places.
2. Representing floating-point numbers in *exponential notation*.
3. Aligning a column of numbers with decimal points appearing one above the other.
4. *Right-justifying* and *left-justifying* outputs.
5. Inserting characters or strings at precise locations in a line of output.
6. Displaying all types of data with fixed-size *field widths* and *precision*.

The program in Fig. 2.19 demonstrates basic string-formatting capabilities.

```
# Fig. 2.19: fig02_19.py
# String formatting.

integerValue = 4237
print "Integer ", integerValue
print "Decimal integer %d" % integerValue
print "Hexadecimal integer %x\n" % integerValue

```

Fig. 2.19 String-formatting operator `%`. (Part 1 of 2.)

```
9  floatValue = 123456.789
10 print "Float", floatValue
11 print "Default float %f" % floatValue
12 print "Default exponential %e\n" % floatValue
13
14 print "Right justify integer (%8d)" % integerValue
15 print "Left justify integer  (%-8d)\n" % integerValue
16
17 stringValue = "String formatting"
18 print "Force eight digits in integer %.8d" % integerValue
19 print "Five digits after decimal in float %.5f" % floatValue
20 print "Fifteen and five characters allowed in string:"
21 print "(%.15s) (%.5s)" % ( stringValue, stringValue )
```

```
Integer  4237
Decimal integer 4237
Hexadecimal integer 108d

Float 123456.789
Default float 123456.789000
Default exponential 1.234568e+005

Right justify integer (    4237)
Left justify \integer  (4237    )

Force eight digits in integer 00004237
Five digits after decimal in float 123456.78900
Fifteen and five characters allowed in string:
(String formatti) (Strin)
```

Fig. 2.19 String-formatting operator **%**. (Part 2 of 2.)

Lines 4–7 demonstrate how to represent integers in a string. Line 5 displays the value of variable **integerValue** without string formatting. The **%** *formatting operator* inserts the value of a variable in a string (line 6). The value to the left of the operator is a string that contains one or more *conversion specifiers*—place holders for values in the string. Each conversion specifier begins with a percent sign (**%**)—not to be confused with the **%** formatting operator—and ends with a *conversion-specifier symbol*. Conversion-specifier symbol **d** indicates that we want to place an integer within the current string at the specified point. Figure 2.20 lists several conversion-specifier symbols for use in string formatting. [*Note:* See Appendix C, Number Systems, for a discussion of numeric terminology in Fig. 2.20.]

Conversion Specifier Symbol	Meaning
c	Single character (i.e., a string of length one) or the integer representation of an ASCII character.
s	String or a value to be converted to a string.

Fig. 2.20 String-formatting characters. (Part 1 of 2.)

Conversion Specifier Symbol	Meaning
d	Signed decimal integer.
u	Unsigned decimal integer.
o	Unsigned octal integer.
x	Unsigned hexadecimal integer (with hexadecimal digits **a** through **f** in lowercase letters).
X	Unsigned hexadecimal integer (with hexadecimal digits **A** through **F** in uppercase letters).
f	Floating-point number.
e, **E**	Floating-point number (using scientific notation).
g, **G**	Floating-point number (using least-significant digits).

Fig. 2.20 String-formatting characters. (Part 2 of 2.)

The value to the right of the **%** formatting operator specifies what replaces the placeholders in the strings. In line 6, we specify the value **integerValue** to replace the **%d** placeholder in the string. Line 7 inserts the hexadecimal representation of the value assigned to variable **integerValue** into the string.

Lines 9–12 demonstrate how to insert floating-point values in a string. The ***f*** conversion specifier acts as a place holder for a floating-point value (line 11). To the right of the **%** formatting operator, we use variable **floatValue** as the value to be displayed. The ***e*** conversion specifier acts as a place holder for a floating-point value in exponential notation. *Exponential notation* is the computer equivalent of scientific notation used in mathematics. For example, the value 150.4582 is represented in *scientific notation* as $\mathbf{1.504582\ X\ 10^2}$ and is represented in exponential notation as **1.504582E+002** by the computer. This notation indicates that **1.504582** is multiplied by **10** raised to the second power (**E+002**). The **E** stands for “exponent.”

Lines 14–15 demonstrate string formatting with *field widths*. A field width is the minimum size of a field in which a value is printed. If the field width is larger than the value being printed, the data is normally *right-justified* within the field. To use field widths, place an integer representing the field width between the percent sign and the conversion-specifier symbol. Line 14 right-justifies the value of variable **integerValue** in a field width of size eight. To *left-justify* a value, specify a negative integer as the field width (line 15).

Lines 17–21 demonstrate string formatting with precision. *Precision* has different meaning for different data types. When used with integer conversion specifiers, precision indicates the minimum number of digits to be printed. If the printed value contains fewer digits than the specified precision, zeros are prefixed to the printed value until the total number of digits is equivalent to the precision. To use precision, place a decimal point (**.**) followed by an integer representing the precision between the percent sign and the conversion specifier. Line 18 prints the value of variable **integerValue** with eight digits of precision.

When precision is used with a floating-point conversion specifier, the precision is the number of digits to appear after the decimal point. Line 19 prints the value of variable **floatValue** with five digits of precision.

When used with a string-conversion specifier, the precision is the maximum number of characters to be written from the string. Line 21 prints the value of variable **stringValue** twice—once with a precision of fifteen and once with a precision of five. Notice that the conversion specifications are contained within parentheses. When the string to the left of the **%** formatting operator contains more than one conversion specifier, the value to the right of the operator must be a comma-separated sequence of values. This sequence is contained within parentheses and must have the same number of values as the string has conversion specifiers. Python constructs the string from left to right by matching a placeholder with the next value specified between parentheses and replacing the formatting character with that value.

Python strings support even more powerful string-formatting capabilities through *string methods*, which we discuss in detail in Chapter 13, Strings Manipulation and Regular Expressions.

2.8 Decision Making: Equality and Relational Operators

This section introduces a simple version of Python's ***if*** *structure* that allows a program to make a decision based on the truth or falsity of some *condition.* If the condition is met, (i.e., the condition is *true),* the statement in the body of the **if** structure is executed. If the condition is not met (i.e., the condition is *false*), the body statement does not execute. We will see an example shortly.

Conditions in **if** structures can be formed with the *equality operators* and *relational operators* summarized in Fig. 2.21. The relational operators all have the same level of precedence and associate from left to right. All equality operators have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate from left to right.

Standard algebraic equality operator or relational operator	Python equality or relational operator	Example of Python condition	Meaning of Python condition
Relational operators			
>	**>**	**x > y**	**x** is greater than **y**
<	**<**	**x < y**	**x** is less than **y**
≥	**>=**	**x >= y**	**x** is greater than or equal to **y**
≤	**<=**	**x <= y**	**x** is less than or equal to **y**
Equality operators			
=	**==**	**x == y**	**x** is equal to **y**
≠	**!=**, **<>**	**x != y**, **x <> y**	**x** is not equal to **y**

Fig. 2.21 Equality and relational operators.

Common Programming Error 2.2

A syntax error occurs if any of the operators `==`, `!=`, `>=` *and* `<=` *appears with spaces between its pair of symbols.*

Common Programming Error 2.3

Reversing the order of the pair of operators in any of the operators `!=`, `<>`, `>=` *and* `<=` *(by writing them as* `=!`, `><`, `=>` *and* `=<`, *respectively) is a syntax error.*

Common Programming Error 2.4

Confusing the equality operator `==` *with the assignment symbol* `=` *is an error. The equality operator should be read "is equal to" and the assignment symbol should be read "gets," "gets the value of" or "is assigned the value of." Some people prefer to read the equality operator as "double equals." In Python, the assignment symbol causes a syntax error when used in a conditional statement.*

The following example uses six **`if`** structures to compare two user-entered numbers. If the condition in any of these **`if`** structures is true, the assignment statement associated with that **`if`** structure executes. The user inputs two values, and the program converts the input values to integers and assigns them to variables **`number1`** and **`number2`**. Then, the program compares the numbers and displays the results of the comparisons. Figure 2.22 shows the program and sample executions.

```python
# Fig. 2.22: fig02_22.py
# Compare integers using if structures, relational operators
# and equality operators.

print "Enter two integers, and I will tell you"
print "the relationships they satisfy."

# read first string and convert to integer
number1 = raw_input( "Please enter first integer: " )
number1 = int( number1 )

# read second string and convert to integer
number2 = raw_input( "Please enter second integer: " )
number2 = int( number2 )

if number1 == number2:
   print "%d is equal to %d" % ( number1, number2 )

if number1 != number2:
   print "%d is not equal to %d" % ( number1, number2 )

if number1 < number2:
   print "%d is less than %d" % ( number1, number2 )

if number1 > number2:
   print "%d is greater than %d" % ( number1, number2 )

```

Fig. 2.22 Equality and relational operators used to determine logical relationships. (Part 1 of 2.)

```
28 if number1 <= number2:
29    print "%d is less than or equal to %d" % ( number1, number2 )
30
31 if number1 >= number2:
32    print "%d is greater than or equal to %d" % ( number1, number2 )
```

```
Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 37
Please enter second integer: 42
37 is not equal to 42
37 is less than 42
37 is less than or equal to 42
```

```
Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 7
Please enter second integer: 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 54
Please enter second integer: 17
54 is not equal to 17
54 is greater than 17
54 is greater than or equal to 17
```

Fig. 2.22 Equality and relational operators used to determine logical relationships. (Part 2 of 2.)

The program uses Python functions **raw_input** and **int** to input two integers (lines 8–14). First a value is obtained for variable **number1**, then a value is obtained for variable **number2**.

The **if** structure in lines 16–17 compares the values of variables **number1** and **number2** to test for equality. If the values are equal, the statement displays a line of text indicating that the numbers are equal (line 17). If the conditions are met in one or more of the **if** structures starting at lines 19, 22, 25, 28 and 31, the corresponding **print** statement displays a line of text.

Each **if** structure consists of the word **if**, the condition to be tested and a colon (**:**). An **if** structure also contains a body (called a *suite*). Notice that each **if** structure in Fig. 2.22 has a single statement in its body and that each body is indented. Some languages, like C++, Java and C# use braces, **{ }**, to denote the body of **if** structures; Python requires indentation for this purpose. We discuss indentation in the next section.

Common Programming Error 2.5

Failure to insert a colon (`:`) in an `if` structure is a syntax error.

Common Programming Error 2.6

Failure to indent the body of an `if` structure is a syntax error.

Good Programming Practice 2.8

Set a convention for the size of indent you prefer, then apply that convention uniformly. The tab key may create indents, but tab stops may vary. We recommend using three spaces to form a level of indent.

In Python, syntax evaluation is dependent on white space; thus, the inconsistent use of white space can cause syntax errors. For instance, splitting a statement over multiple lines can result in a syntax error. If a statement is long, the statement can be spread over multiple lines using the \ line-continuation character. Some Python interpreters use `"..."` to denote a continuing line. The interactive session in Fig. 2.23 demonstrates the line-continuation character.

Good Programming Practice 2.9

A lengthy statement may be spread over several lines with the \ continuation character. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a `print` statement or after an operator in a lengthy expression.

Figure 2.24 shows the precedence of the operators introduced in this chapter. The operators are shown from top to bottom in decreasing order of precedence. Notice that all these operators, except exponentiation, associate from left to right.

Testing and Debugging Tip 2.3

Refer to the operator-precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order, exactly as you would do in an algebraic expression. Be sure to observe that some operators, such as exponentiation (`**`), associate from right to left rather than from left to right.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> print 1 +
  File "<string>", line 1
    print 1 +
             ^
SyntaxError: invalid syntax
>>> print 1 + \
... 2
3
>>>
```

Fig. 2.23 Line-continuation (\) character.

Operators	Associativity	Type
`()`	left to right	parentheses
`**`	right to left	exponential
`*  /  //  %`	left to right	multiplicative
`+  -`	left to right	additive
`<  <=  >  >=`	left to right	relational
`==  !=  <>`	left to right	equality

Fig. 2.24 Precedence and associativity of operators discussed so far.

2.9 Indentation

Python uses indentation to delimit (distinguish) sections of code. Other programming languages often use braces to delimit sections of code. A *suite* is a section of code that corresponds to the body of a control structure. We study *blocks* in the next chapter. The Python programmer chooses the number of spaces to indent a suite or block, and the number of spaces must remain consistent for each statement in the suite or block. Python recognizes new suites or blocks when there is a change in the number of indented spaces.

Common Programming Error 2.7

If a single section of code contains lines of code that are not uniformly indented, the Python interpreter reads those lines as belonging to other sections, causing syntax or logic errors.

Figure 2.25 contains a modified version of the code in Fig. 2.22 to illustrate improper indentation. Lines 21–22 show the improper indentation of an **`if`** statement. Even though the program does not produce an error, it skips an equality operator. The

```
if number1 != number2:
```

statement (line 21) executes only if the **`if number1 == number2:`** statement (line 16) executes. In this case, the **`if`** statement in line 21 never executes, because two equal numbers will never be unequal (i.e., 2 will never unequal 2). Thus, the output of Fig. 2.25 does not state that **`1 is not equal to 2`** as it should.

```
1  # Fig. 2.25: fig02_25.py
2  # Using if statements, relational operators and equality
3  # operators to show improper indentation.
4
5  print "Enter two integers, and I will tell you"
6  print "the relationships they satisfy."
7
8  # read first string and convert to integer
9  number1 = raw_input( "Please enter first integer: " )
10 number1 = int( number1 )
11
```

Fig. 2.25 **`if`** statements used to show improper indentation. (Part 1 of 2.)

```
# read second string and convert to integer
number2 = raw_input( "Please enter second integer: " )
number2 = int( number2 )

if number1 == number2:
   print "%d is equal to %d" % ( number1, number2 )

   # improper indentation causes this if statement to execute only
   # when the above if statement executes
   if number1 != number2:
      print "%d is not equal to %d" % ( number1, number2 )

if number1 < number2:
   print "%d is less than %d" % ( number1, number2 )

if number1 > number2:
   print "%d is greater than %d" % ( number1, number2 )

if number1 <= number2:
   print "%d is less than or equal to %d" % ( number1, number2 )

if number1 >= number2:
   print "%d is greater than or equal to %d" % ( number1, number2 )
```

```
Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 1
Please enter second integer: 2
1 is less than 2
1 is less than or equal to 2
```

Fig. 2.25 `if` statements used to show improper indentation. (Part 2 of 2.)

Testing and Debugging Tip 2.4

To avoid subtle errors, ensure consistent and proper indentation within a Python program.

2.10 Thinking About Objects: Introduction to Object Technology

In each of the first six chapters, we concentrate on the "conventional" methodology of structured programming, because the objects we will build will be composed in part of structured-program pieces. Now we begin our early introduction to object orientation. In this section, we will see that object orientation is a natural way of thinking about the world and of writing computer programs.

We begin our introduction to object orientation with some key concepts and terminology. First, look around you in the real world. Everywhere you look you see them—*objects*!—people, animals, plants, cars, planes, buildings, computers, etc. Humans think in terms of objects. We have the marvelous ability of *abstraction* that enables us to view

images on a computer screen as objects such as people, planes, trees and mountains, rather than as individual dots of color. We can, if we wish, think in terms of beaches rather than grains of sand, forests rather than trees and buildings rather than bricks.

We might be inclined to divide objects into two categories—animate objects and inanimate objects. Animate objects are "alive" in some sense. They move around and do things. Inanimate objects, like towels, seem not to do much at all. They just "sit around." All these objects, however, do have some things in common. They all have *attributes*, like size, shape, color and weight, and they all exhibit *behaviors* (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water).

Humans learn about objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees. Cars, trucks, little red wagons and roller skates have much in common.

Object-oriented programming (OOP) models real-world objects using software counterparts. It takes advantage of *class* relationships, where objects of a certain class—such as a class of vehicles—have the same characteristics. It takes advantage of *inheritance* relationships, and even *multiple inheritance* relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class "convertible" certainly has the characteristics of the more general class "automobile," but a convertible's roof goes up and down.

Object-oriented programming gives us a more natural and intuitive way to view the programming process, by *modeling* real-world objects, their attributes and their behaviors. OOP also models communications between objects. Just as people send *messages* to one another (e.g., a sergeant commanding a soldier to stand at attention), objects communicate via messages.

OOP *encapsulates* data (attributes) and functions (behavior) into packages called *objects;* the data and functions of an object are intimately tied together. Objects have the property of *information hiding.* This means that, although objects may know how to communicate with one another, objects normally are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. Surely it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is so crucial to good software engineering.

In C and other *procedural programming languages,* programming tends to be *action-oriented;* in Python, programming is *object-oriented* (ideally). The function is the unit of programming in procedural programming. In object-oriented programming, the unit of programming is the *class* from which objects are eventually *instantiated* (a fancy term for "created"). Python classes contain functions (that implement class behaviors) and data (that implements class attributes).

Procedural programmers concentrate on writing functions. Groups of actions that perform some task are formed into functions, and functions are grouped to form programs. Data is certainly important in procedural programming, but the view is that data exists primarily in support of the actions that functions perform. The verbs in a system specification help the procedural programmer determine the set of functions that will work together to implement the system.

Object-oriented programmers concentrate on creating their own *user-defined types* called *classes*. Each class contains both data and the set of functions that manipulate the data. The data components of a class are called *data members* or *attributes*. The functional components of a class are called *methods* (or *member functions* in other object-oriented languages). The focus of attention in object-oriented programming is on classes rather than functions. The *nouns* in a system specification help the object-oriented programmer determine the set of classes that will be used to create the instances that will work together to implement the system.

Classes are to objects as blueprints are to houses. We can build many houses from one blueprint, and we can create many objects from one class. Classes can also have relationships with other classes. For example, in an object-oriented design of a bank, the **`BankTeller`** class needs to relate to the **`Customer`** class. These relationships are called *associations*.

We will see that, when software is packaged as classes, these classes can be *reused* in future software systems. Groups of related classes are often packaged as reusable *components* or *modules*. Just as real-estate brokers tell their clients that the three most important factors affecting the price of real estate are "location, location and location," we believe the three most important factors affecting the future of software development are "reuse, reuse and reuse."

Indeed, with object technology, we will build most future software by combining "standardized, interchangeable parts" called components. This book will teach you how to "craft valuable classes" for reuse, reuse and reuse. Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software-development efforts. This is an exciting possibility.

In this chapter, we have introduced many important features of Python, including printing data on the screen, inputting data from the keyboard, performing calculations and making decisions. In Chapter 3, Control Structures, we build on these techniques as we introduce *structured programming*. We will study how to specify and vary the order in which statements are executed—this order is called *flow of control*. Also, we introduced the basic concepts and terminology of object orientation. In Chapters 7–9, we expand our discussion on object-oriented programming.

SUMMARY

- Programmers insert comments to document programs and to improve program readability. Comments also help other programmers read and understand your program. In Python, comments are denoted by the pound symbol (**`#`**).
- A comment that begins with **`#`** is called a single-line comment, because the comment terminates at the end of the current line.
- Comments do not cause the computer to perform any action when the program is run. Python ignores comments.
- Programmers use blank lines and space characters to make programs easier to read. Together, blank lines, space characters and tab characters are known as white space. (Space characters and tabs are known specifically as white-space characters.)
- Blank lines are ignored by Python.
- The standard output stream is the channel by which information presented to the user by an application—this information typically is displayed on the screen, but may be printed on a printer, writ-

ten to a file, etc. It may even be spoken or issued to braille devices, so users with visual impairments can receive the outputs.

- The **print** statement instructs the computer to display the string of characters contained between the quotation marks. A string is a Python data type that contains a sequence of characters.
- A **print** statement normally sends a newline character to the screen. After a newline character is sent, the next string displayed on the screen appears on the line below the previous string. However, a comma (**,**) tells Python not to send the newline character to the screen. Instead, Python adds a space after the string, and the next string printed to the screen appears on the same line.
- Output (i.e., displaying information) and input (i.e., receiving information) in Python are accomplished with streams of characters.
- Python files typically end with **.py**, although other extensions (e.g., **.pyw** on Windows) can be used.
- When the Python interpreter executes a program, the interpreter starts at the first line of the file and executes statements until the end of the file.
- The backslash (****) is an escape character. It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence.
- The escape sequence **\n** means newline. Each occurrence of a **\n** (newline) escape sequence causes the screen cursor to position to the beginning of the next line.
- A built-in function is a piece of code provided by Python that performs a task. The task is performed when the function is invoked or called. After performing its task, a function may return a value that represents the end result of the task.
- In Python, variables are more specifically referred to as objects. An object resides in the computer's memory and contains information used by the program. The term object normally implies that attributes (data) and behaviors (methods) are associated with the object. The object's methods use the attributes to perform tasks.
- A variable name consists of letters, digits and underscores (_) and does not begin with a digit.
- Python is case sensitive—uppercase and lowercase letters are different, so **a1** and **A1** are different variables.
- An object can have multiple names, called identifiers. Each identifier (or variable name) references (points to) the object (or variable) in memory.
- Each object has a type. An object's type identifies the kind of information (e.g., integer, string, etc.) stored in the object.
- In Python, every object has a type, a size, a value and a location.
- Function **type** returns the type of an object. Function **id** returns a number that represents the object's location.
- In languages like C++ and Java, the programmer must declare the object type before using the object in the program. In Python, the type of an object is determined automatically, as the program executes. This approach is called dynamic typing.
- Binary operators take two operands. Examples of binary operators are **+** and **-**.
- Starting with Python version 2.2, the behavior of the **/** division operator will change from "floor division" to "true division."
- Floor division (sometimes called integer division), divides the numerator by the denominator and returns the highest integer value that is not greater than the result. Any fractional part in floor division is simply discarded (i.e., truncated)—no rounding occurs.

- True division yields the precise floating-point result of dividing the numerator by the denominator.
- The behavior (i.e., floor or true division) of the **/** operator is determined by the type of the operands. If the operands are both integers, the operator performs floor division. If one or both of the operands are floating-point numbers, the operator perform true division.
- The **//** operator performs floor division.
- Programmers can change the behavior of the **/** operator to perform true division with the statement **from __future__ import division**.
- In Python version 3.0, the only behavior of the **/** operator will be true division. After the release of version 3.0, all programs are expected to have been updated to compensate for the new behavior.
- Python provides the modulus operator (**%**), which yields the remainder after integer division. The expression **x % y** yields the remainder after **x** is divided by **y**. Thus, **7 % 4** yields **3** and **17 % 5** yields **2**. This operator is most commonly used with integer operands, but also can be used with other arithmetic types.
- The modulus operator can be used with both integer and floating-point numbers.
- Arithmetic expressions in Python must be entered into the computer in straight-line form. Thus, expressions such as "**a** divided by **b**" must be written as **a / b**, so that all constants, variables and operators appear in a straight line.
- Parentheses are used in Python expressions in much the same manner as in algebraic expressions. For example, to multiply **a** times the quantity **b + c**, we write **a * (b + c)**.
- Python applies operators in arithmetic expressions in a precise sequence determined by the rules of operator precedence, which are generally the same as those followed in algebra.
- When we say that certain operators are applied from left to right, we are referring to the associativity of the operators.
- Python provides strings as a built-in data type and can perform powerful text-based operations.
- Strings can be created using the single-quote (**'**) and double-quote characters (**"**). Python also supports triple-quoted strings. Triple-quoted strings are useful for programs that output strings with quote characters or large blocks of text. Single- or double-quote characters inside a triple-quoted string do not need to use the escape sequence, and triple-quoted strings can span multiple lines.
- A field width is the minimum size of a field in which a value is printed. If the field width is larger than that needed by the value being printed, the data normally is right-justified within the field. To use field widths, place an integer representing the field width between the percent sign and the conversion-specifier symbol.
- Precision has different meaning for different data types. When used with integer conversion specifiers, precision indicates the minimum number of digits to be printed. If the printed value contains fewer digits than the specified precision, zeros are prefixed to the printed value until the total number of digits is equivalent to the precision.
- When used with a floating-point conversion specifier, the precision is the number of digits to appear to the right of the decimal point.
- When used with a string-conversion specifier, the precision is the maximum number of characters to be written from the string.
- Exponential notation is the computer equivalent of scientific notation used in mathematics. For example, the value 150.4582 is represented in scientific notation as 1.504582×10^2 and is represented in exponential notation as **1.504582E+002** by the computer. This notation indicates that **1.504582** is multiplied by **10** raised to the second power (**E+002**). The **E** stands for "exponent."

- An **`if`** structure allows a program to make a decision based on the truth or falsity of a condition. If the condition is true, (i.e., the condition is met), the statement in the body of the **`if`** structure is executed. If the condition is not met, the body statement is not executed.
- Conditions in **`if`** structures can be formed with equality relational operators. The relational operators all have the same level of precedence and associate from left to right. The equality operators both have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate from left to right.
- Each **`if`** structure consists of the word **`if`**, the condition to be tested and a colon (**`:`**). An **`if`** structure also contains a body (called a suite).
- Python uses indentation to delimit (distinguish) sections of code. Other programming languages often use braces to delimit sections of code. A suite is a section of code that corresponds to the body of a control structure. We study blocks in the next chapter.
- The Python programmer chooses the number of spaces to indent a suite or block, and the number of spaces must remain consistent for each statement in the suite or block.
- Splitting a statement over two lines can also cause a syntax error. If a statement is long, the statement can be spread over multiple lines using the **`\`** line-continuation character.
- Object-oriented programming (OOP) models real-world objects with software counterparts. It takes advantage of class relationships where objects of a certain class—such as a class of vehicles—have the same characteristics.
- OOP takes advantage of inheritance relationships, and even multiple-inheritance relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- Object-oriented programming gives us a more natural and intuitive way to view the programming process, namely, by modeling real-world objects, their attributes and their behaviors. OOP also models communication between objects.
- OOP encapsulates data (attributes) and functions (behavior) into packages called objects; the data and functions of an object are intimately tied together.
- Objects have the property of information hiding. Although objects may know how to communicate with one another across well-defined interfaces, objects normally are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves.
- In Python, programming can be object-oriented. In object-oriented programming, the unit of programming is the class from which instances are eventually created. Python classes contain methods (that implement class behaviors) and data (that implements class attributes).
- Object-oriented programmers create their own user-defined types called classes and components. Each class contains both data and the set of functions that manipulate the data. The data components of a class are called data members or attributes.
- The functional components of a class are called methods (or member functions, in some other object-oriented languages).
- The focus of attention in object-oriented programming is on classes rather than on functions. The nouns in a system specification help the object-oriented programmer determine the set of classes that will be used to create the instances that will work together to implement the system.

TERMINOLOGY

abstraction
alert escape sequence (**`\a`**)
argument
arithmetic operator
assignment statement
assignment symbol (**`=`**)

association
associativity
associativity of operators
asterisk (`*`)
attribute
backslash (`\`) escape sequence
backspace (`\b`)
behavior
binary operator
block
built-in function
calculation
calling a function
carriage return (`\r`)
case sensitive
class
comma-separated list
comment
component
condition
conversion specifier
data member
debugging
design
dynamic typing
embedded parentheses
encapsulation
equality operators
escape character
escape sequence
execute
exponential notation
exponentiation
field width
floating-point division
floor division
flow of control
function
`id` function
identifier
indentation
information hiding
inheritance
instance
`int` function
integer division
left justify
left-to-right evaluation
member function
memory
memory location
method
modeling
modulus
modulus operator (`%`)
multiple inheritance
newline character (`\n`)
object
object orientation
OOP (object-oriented programming)
operand
operator overloading
operator precedence
overloading
percent sign (`%`)
polynomial
precedence
precision
procedural programming language
pseudocode
`.py` extension
`.pyw` extension
`raw_input` function
readability
redundant parentheses
relational operator
reused class
right justify
scientific notation
screen output
second-degree polynomial
self-documentation
single-line comment
single quote
software asset
standard output stream
statement
stream of characters
string of characters
string type
structured programming
suite
system path variable
triple-quoted string
true division
truncate
type
`type` function
user-defined type
variable

SELF-REVIEW EXERCISES

2.1 Fill in the blanks in each of the following:

a) The ________ statement instructs the computer to display information on the screen.
b) A ________ is a Python data type that contains a sequence of characters.
c) ________ are simply names that reference objects.
d) The ________ is the modulus operator.
e) ________ are used to document a program and improve its readability.
f) Each **`if`** structure consists of the word ________, the ________ to be tested, a ________ and a ________.
g) The ________ function converts non-integer values to integer values.
h) A Python statement can be spread over multiple lines using the ________.
i) Arithmetic expressions enclosed in ________ are evaluated first.
j) An object's ________ describes the information stored in the object.

2.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The Python function **`get_input`** requests input from the user.
b) A valid Python arithmetic expression with no parentheses is evaluated left to right.
c) The following are invalid variable names: **`3g`**, **`87`** and **`2h`**.
d) The operator **`!=`** is an example of a relational operator.
e) A variable name identifies the kind of information stored in the object.
f) In Python, the programmer must declare the object type before using the object in the program.
g) If parentheses are nested, the expression in the innermost pair is evaluated first.
h) Python treats the variable names, **`a1`** and **`A1`**, as the same variable.
i) The backslash character is called an escape sequence.
j) The relational operators all have the same level of precedence and evaluate left to right.

ANSWERS TO SELF-REVIEW EXERCISES

2.1 a) **`print`**. b) string. c) Identifiers. d) percent sign (**`%`**). e) Comments. f) **`if`**, condition, colon (**`:`**), body/suite. g) **`int`**. h) line-continuation character (**`\`**). i) parentheses. j) type.

2.2 a) False. The Python function **`raw_input`** gets input from the user. b) False. Python arithmetic expressions are evaluated according to the rules of operator precedence and associativity—not left to right. c) True. d) False. The operator **`!=`** is an example of an equality operator. e) False. An object type identifies the kind of information stored in the object. f) False. In Python, the object type is determined as the program executes. g) True. h) False. Python is case sensitive, so **`a1`** and **`A1`** are different variables. i) False. The backslash is called an escape character. j) True.

EXERCISES

2.3 State the order of evaluation of the operators in each of the following Python statements and show the value of **`x`** after each statement is performed.

```
a) x = 7 + 3 * 6 / 2 - 1
b) x = 2 % 2 + 2 * 2 - 2 / 2
c) x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) )
```

2.4 Write a program that requests the user to enter two numbers and prints the sum, product, difference and quotient of the two numbers.

2.5 Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for π. Do these calculations in output statements.

2.6 Write a program that prints a box, an oval, an arrow and a diamond, as shown:

```
*********          ***              *              *
*       *        *     *           ***            * *
*       *       *       *         *****          *   *
*       *       *       *           *           *     *
*       *       *       *           *          *       *
*       *       *       *           *           *     *
*       *       *       *           *            *   *
*       *        *     *            *             * *
*********           ***               *              *
```

2.7 Write a program that reads in two integers and determines and prints whether the first is a multiple of the second. (Hint: Use the modulus operator.)

2.8 Give a brief answer to each of the following "object think" questions:

a) Why does this text choose to discuss structured programming in detail before proceeding with an in-depth treatment of object-oriented programming?

b) What aspects of an object need to be determined before an object-oriented program can be built?

c) How is inheritance exhibited by human beings?

d) What kinds of messages do people send to one another?

e) Objects send messages to one another across well-defined interfaces. What interfaces does a car radio (object) present to its user (a person object)?

3

Control Structures

Objectives

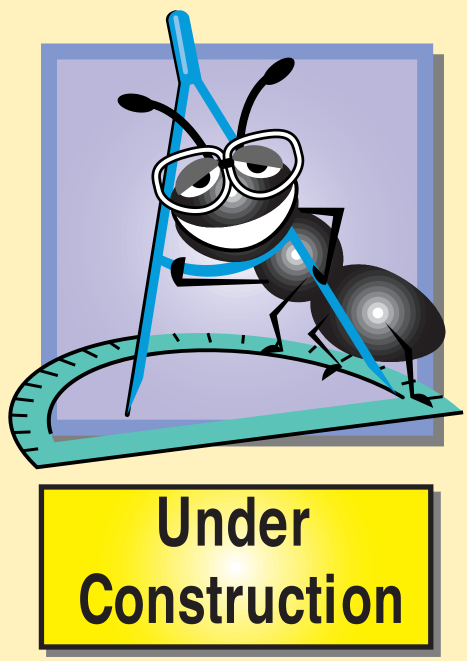
- To understand basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the **`if`**, **`if`**/**`else`** and **`if`**/**`elif`**/**`else`** structures to select appropriate actions.
- To use the **`while`** and **`for`** repetition structures to execute statements in a program repeatedly.
- To understand counter-controlled and sentinel-controlled repetition.
- To use augmented assignment symbols and logical operators.
- To use the **`break`** and **`continue`** program control statements.

Let's all move one place on.
Lewis Carroll

The wheel is come full circle.
William Shakespeare, King Lear

Who can control his fate?
William Shakespeare, Othello

The used key is always bright.
Benjamin Franklin



Outline

3.1 Introduction

Before writing a program to solve a particular problem, it is essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem. When writing a program, it is equally essential to understand the types of building blocks that are available and to use proven program-construction principles. In this chapter, we discuss these issues in our presentation of the theory and principles of structured programming. The techniques that you learn are applicable to most high-level languages, including Python. When we begin our treatment of object-oriented programming in Chapter 7, we use the control structures presented in this chapter to build and manipulate objects.

3.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specified order. An *algorithm* is a *procedure* for solving a problem in terms of

1. *actions* to be executed and
2. the *order* in which these actions are to be executed.

The following example demonstrates that specifying the order in which the actions are to be executed is important.

Consider the "rise-and-shine" algorithm followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work. This routine gets the executive to work to make critical decisions.

Suppose that the same steps are performed in a slightly different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our junior executive shows up for work soaking wet.

Specifying the order in which statements are to be executed in a computer program is called *program control*. In this chapter, we investigate Python's program-control capabilities.

3.3 Pseudocode

Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode consists of descriptions of *executable statements*—those that are executed when the program has been converted from pseudocode to Python. The pseudocode we present here is useful for developing algorithms that will be converted to Python programs. Pseudocode is similar to everyday English; it is convenient and user-friendly, although it is not an actual computer programming language.

Pseudocode programs are not executed on computers. Rather, pseudocode helps the programmer "plan" a program before attempting to write it in a programming language, such as Python. In this chapter, we provide several examples of how pseudocode can be used effectively in developing Python programs.

Software Engineering Observation 3.1

Pseudocode often is used to "think out" a program during the program design process. Then the pseudocode program is converted to Python.

The style of pseudocode we present consists purely of characters, so programmers can conveniently type pseudocode programs using a text-editor program. This way, a computer can display a fresh copy of a pseudocode program on demand. A carefully prepared pseudocode program can be converted easily to a corresponding Python program. In many cases, this is done simply by replacing pseudocode statements with their Python equivalents.

3.4 Control Structures

Normally, statements in a program are executed in the order in which they are written. This is called *sequential execution*. Various Python statements enable the programmer to specify that the next statement to be executed may be other than the next one in sequence. This is called *transfer of control.* Transfer of control is achieved with Python *control structures.* This section discusses the background of control structure development and the specific tools Python uses to transfer control in a program.

During the 1960s, it became clear that the indiscriminate use of control transfers caused the difficulty experienced by software-development groups. The finger of blame was pointed at the **`goto`** *statement* (used in several programming languages, including C and Basic), which allows a programmer to specify a transfer of control to one of a wide range of possible destinations in a program. The notion of so-called *structured programming* became almost synonymous with "goto elimination."

The research of Bohm and Jacopini[1] demonstrated that programs could be written without any **goto** statements. The challenge, then became for programmers to alter their programming styles to "**goto**-less programming." When programmers began to take structured programming seriously beginning in the 1970s, the notion of structured programming became almost synonymous with goto elimination. Since then, the results have been impressive, as software development groups have reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. Structured programming has enabled these improvements because structured programs are clearer, easier to debug and modify and more likely to be bug-free in the first place.

Bohm and Jacopini's work demonstrated that all programs could be written in terms of three *control structures*—namely, the *sequence structure*, the *selection structure* and the *repetition structure*. The sequence structure is built into Python. Unless directed otherwise, the computer executes Python statements sequentially. The *flowchart* segment of Fig. 3.1 illustrates a typical sequence structure in which two calculations are performed sequentially. A flowchart is a tool that provides graphical representation of an algorithm or a portion of an algorithm.

Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called *flowlines*, which indicate the order in which the actions of the algorithm execute. Like pseudocode, flowcharts aid in the development and representation of algorithms. Although most programmers prefer pseudocode, flowcharts illustrate clearly how control structures operate. The reader should carefully compare the pseudocode and flowchart representations of each control structure.

The flowchart segment for the sequence structure in Fig. 3.1 uses the *rectangle* symbol, called the *action* symbol, to indicate an action, (e.g., calculation or an input/output operation). The flowlines in the figure indicate the order in which the actions are to be performed—first, **grade** is added to **total**, then **1** is added to **counter**. Python allows us to have as many actions as we want in a sequence structure—anywhere a single action may be placed, we can place several actions in sequence.

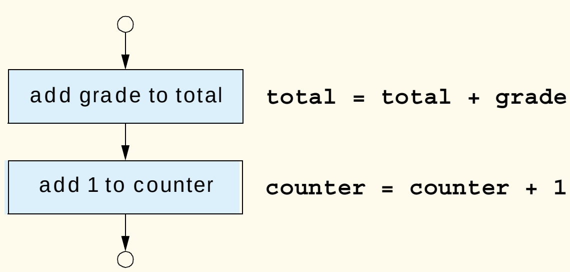


Fig. 3.1 Sequence structure flowchart.

1. Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

In a flowchart that represents a *complete* algorithm, an *oval* symbol containing the word "Begin" represents the start of the flowchart; an oval symbol containing the word "End" represents the end of the flowchart. When drawing a portion of an algorithm, as in Fig. 3.1, the oval symbols are omitted in favor of *small circle* symbols, also called *connector* symbols.

Perhaps the most important flowchart symbol is the *diamond* symbol, also called the *decision* symbol, which indicates a decision is to be made. We discuss the diamond symbol in the next section. The pseudocode we present here is useful for developing algorithms that will be converted to structured Python programs.

Python provides three types of selection structures: **`if`**, **`if`**/**`else`** and **`if`**/**`elif`**/**`else`**. We discuss each of these in this chapter. The **`if`** selection structure either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false. The **`if`**/**`else`** selection structure performs an action if a condition is true or performs a different action if the condition is false. The **`if`**/**`elif`**/**`else`** selection structure performs one of many different actions, depending on the truth or falsity of several conditions.

The **`if`** selection structure is a *single-selection* structure because it selects or ignores a single action. The **`if`**/**`else`** selection structure is a *double-selection* structure because it selects between two different actions. The **`if`**/**`elif`**/**`else`** selection structure is a *multiple-selection* structure because it selects the action to perform from many different actions.

Python provides two types of repetition structures: **`while`** and **`for`**. The **`if`**, **`elif`**, **`else`**, **`while`** and **`for`** structures are Python *keywords*. These keywords are reserved by the language to implement various Python features, such as control structures. Keywords cannot be used as identifiers (i.e., variable names). Figure 3.2 lists all Python keywords.[2]

Common Programming Error 3.1

Using a keyword as an identifier is a syntax error.

In all, Python has only the six control structures: the sequence structure, three types of selection structures and two types of repetition structures. Each Python program is formed by combining as many control structures as is appropriate for the algorithm the program implements. As with the sequence structure shown in Fig. 3.1, we will see that each control structure is flowcharted with two small circle symbols, one at the entry point to the control structure and one at the exit point.

Python keywords						
`and`	`continue`	`else`	`for`	`import`	`not`	`raise`
`assert`	`def`	`except`	`from`	`in`	`or`	`return`
`break`	`del`	`exec`	`global`	`is`	`pass`	`try`
`class`	`elif`	`finally`	`if`	`lambda`	`print`	`while`

Fig. 3.2 Python keywords.

2. Python 2.3 will introduce the keyword **`yield`** among others. Visit the Python Web site (**`www.python.org`**) to view a tentative list of such keywords, and avoid using them as identifiers.

These *single-entry/single-exit control structures* make it easy to build programs. The control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This is similar to the way a child stacks building blocks; hence, the term *control-structure stacking*. *Control-structure nesting* also connects control structures; we discuss this technique later in the chapter.

Software Engineering Observation 3.2

Any Python program can be constructed from six different types of control structures (sequence, ***`if`***, ***`if`***/***`else`***, ***`if`***/***`elif`***/***`else`***, ***`while`*** *and* ***`for`****) combined in two ways (control-structure stacking and control-structure nesting).*

3.5 `if` Selection Structure

Selection structures choose among alternative courses of action. For example, suppose that the passing grade on an examination is 60. Then the pseudocode statement

If student's grade is greater than or equal to 60
Print "Passed"

determines whether the condition "student's grade is greater than or equal to 60" is true or false. If the condition is true, then "Passed" is printed, and the next pseudocode statement in order is "performed." (Remember that pseudocode is not a real programming language.) If the condition is false, the print statement is ignored, and the next pseudocode statement is performed. Note that the second line of this selection structure is indented. Such indentation is optional (for pseudocode), but it is highly recommended because indentation emphasizes the inherent hierarchy of structured programs. When we convert pseudocode into Python code, indentation is required.

The preceding pseudocode *if* statement may be written in Python as

```
if grade >= 60:
   print "Passed"
```

Notice that the Python code corresponds closely to the pseudocode. This similarity is the reason that pseudocode is a useful program development tool. The statement in the body of the **`if`** structure outputs the character string **`"Passed"`**.

The flowchart of Fig. 3.3 illustrates the single-selection **`if`** structure and the diamond symbol. The decision symbol contains an expression, such as a condition, that can be either true or false. The diamond has two flowlines emerging from it: One indicates the direction to follow when the expression in the symbol is true; the other indicates the direction to follow when the expression is false. We learned, in Chapter 2, Introduction to Python Programming, that decisions can be based on conditions containing relational or equality operators. Actually, a decision can be based on any expression. For instance, if an expression evaluates to zero, it is treated as false, and if an expression evaluates to nonzero, it is treated as true.

Note that the **`if`** structure is a single-entry/single-exit structure. We will soon learn that the flowcharts for the remaining control structures also contain (besides small circle symbols and flowlines) rectangle symbols that indicate the actions to be performed and diamond symbols that indicate decisions to be made. This type of flowchart emphasizes the *action/decision model of programming*.

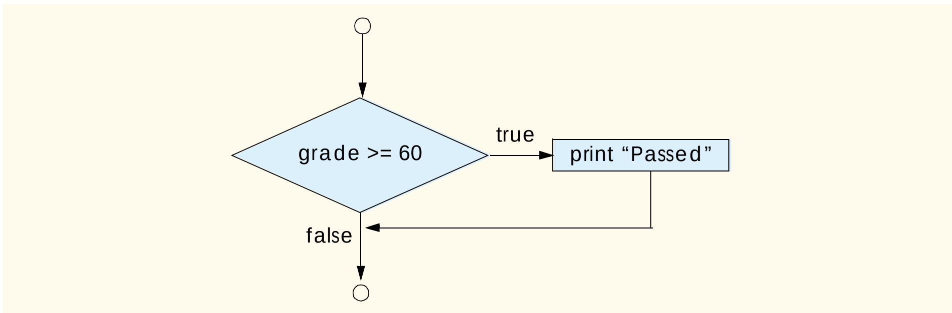


Fig. 3.3 `if` single-selection structure flowchart.

We can envision six bins, each containing control structures of one of the six types. These control structures are empty—nothing is written in the rectangles or in the diamonds. The programmer's task, then, is assembling a program from as many of each type of control structure as the algorithm demands, combining those control structures in only two possible ways (stacking or nesting), then filling in the actions and decisions in a manner appropriate for the algorithm. We will discuss the variety of ways in which actions and decisions may be written.

3.6 `if`/`else` and `if`/`elif`/`else` Selection Structures

The **`if`** selection structure performs a specified action only when the condition is true; otherwise, the action is skipped. The **`if`**/**`else`** selection structure allows the programmer to specify that a different action is to be performed when a condition is true from an action when a condition is false. For example, the pseudocode statement

If student's grade is greater than or equal to 60
    Print "Passed"
else
    Print "Failed"

prints *Passed* if the student's grade is greater than or equal to 60 and prints *Failed* if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed." Note that the body of the *else* is indented. The indented body of a control structure is called a *suite*. Remember that indentation conventions you choose should be applied uniformly throughout programs. It is imperative for Python when it is executing code, and programs that do not obey uniform spacing conventions also are difficult to read.

Good Programming Practice 3.1

If there are several levels of indentation, each suite must be indented. Different suites at the same level do not have to be indented by the same amount, but doing so is good programming practice.

The preceding pseudocode *if/else* structure can be written in Python as

```
if grade >= 60:
   print "Passed"
else:
   print "Failed"
```

Common Programming Error 3.2

Failure to indent all statements that belong to an **`if`** *suite or an* **`else`** *suite results in a syntax error.*

The flowchart of Fig. 3.4 illustrates the flow of control in the **`if`**/**`else`** structure. Once again, note that (besides small circles and arrows) the symbols in the flowchart are rectangles (for actions) and diamonds (for decisions). We continue to emphasize this action/decision model of computing. Imagine again a bin containing empty double-selection structures. The programmer's job is to assemble these selection structures (by stacking and nesting) with other control structures required by the algorithm and to fill in the rectangles and diamonds with actions and decisions appropriate to the algorithm being implemented.

Nested **`if`**/**`else`** *structures* test for multiple cases by placing **`if`**/**`else`** selection structures inside other **`if`**/**`else`** selection structures. For example, the following pseudocode statement prints **`A`** for exam grades greater than or equal to 90, **`B`** for grades 80–89, **`C`** for grades 70–79, **`D`** for grades 60–69 and **`F`** for all other grades.

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```

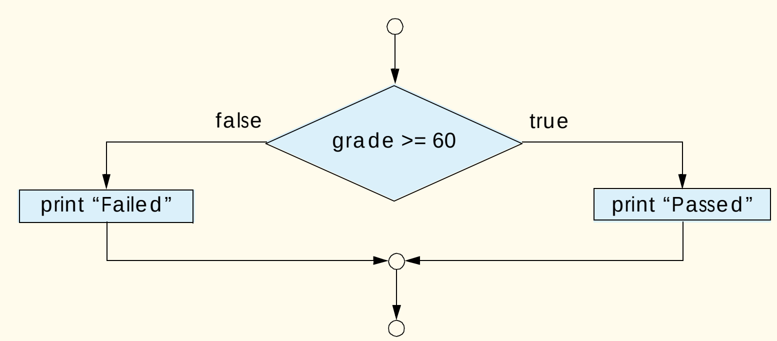


Fig. 3.4 **`if`**/**`else`** double-selection structure flowchart.

This pseudocode can be written in Python as

```
if grade >= 90:
   print "A"
else:
   if grade >= 80:
      print "B"
   else:
      if grade >= 70:
         print "C"
      else:
         if grade >= 60:
            print "D"
         else:
            print "F"
```

If **`grade`** is greater than or equal to 90, the first four conditions are met, but only the **`print`** statement after the first test executes. After that **`print`** executes, the **`else`** part of the “outer” **`if`**/**`else`** statement skips.

Performance Tip 3.1

*A nested **`if`**/**`else`** structure is faster than a series of single-selection **`if`** structures because the testing of conditions terminates after one of the conditions is satisfied.*

Performance Tip 3.2

*In a nested **`if`**/**`else`** structure, place the conditions that are more likely to be true at the beginning of the nested **`if`**/**`else`** structure. This enables the nested **`if`**/**`else`** structure to run faster and exit earlier than an equivalent **`if`**/**`else`** structure in which infrequent cases appear first.*

Many Python programmers prefer to write the preceding **`if`** structure as

```
if grade >= 90:
   print "A"
elif grade >= 80:
   print "B"
elif grade >= 70:
   print "C"
elif grade >= 60:
   print "D"
else:
   print "F"
```

thus replacing the double-selection **`if`**/**`else`** structure with the *multiple-selection* **`if`**/**`elif`**/**`else`** *structure*. The two forms are equivalent. The latter form is popular because it avoids the deep indentation of the code to the right. Such indentation often leaves little room on a line, forcing lines to be split over multiple lines and decreasing program readability.

Each **`elif`** can have one or more actions. The flowchart in Fig. 3.5 shows the general **`if`**/**`elif`**/**`else`** multiple-selection structure. The flowchart indicates that, after an **`if`** or **`elif`** statement executes, control immediately exits the **`if`**/**`elif`**/**`else`** structure. Again, note that (besides small circles and arrows) the flowchart contains rectangle symbols and diamond symbols. Imagine that the programmer has access to a deep bin of empty **`if`**/**`elif`**/**`else`** structures—as many as the programmer might need to stack and nest with

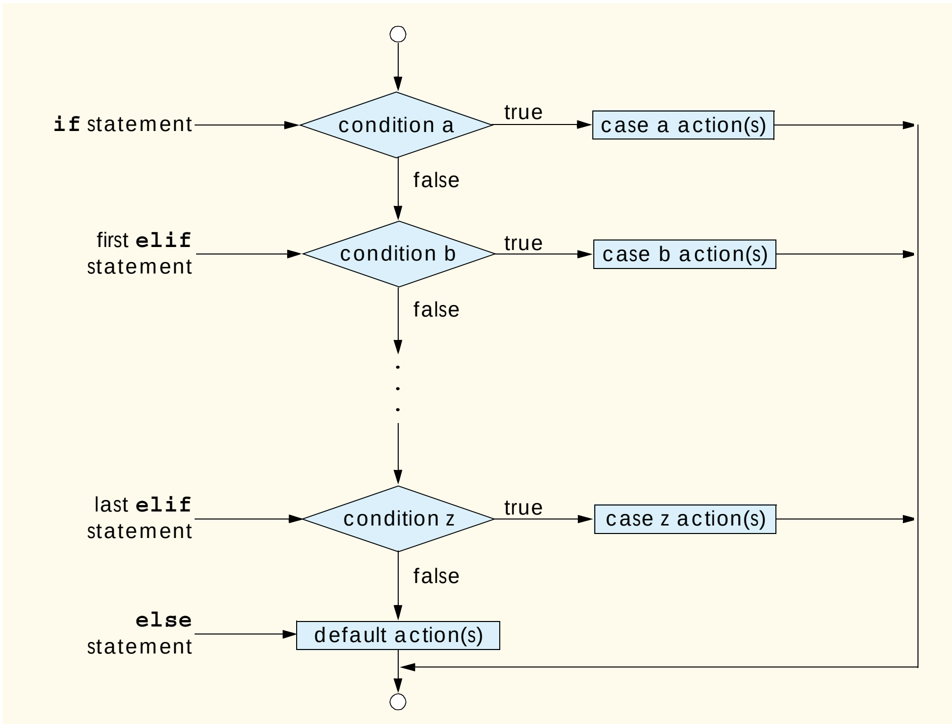


Fig. 3.5 **`if`/`elif`/`else`** multiple-selection structure.

other control structures to form a structured implementation of an algorithm's flow of control. The rectangles and diamonds are then filled with actions and decisions appropriate to the algorithm.

The **`else`** statement of the ***`if`/`elif`/`else`*** structure is optional. However, most programmers include an **`else`** statement at the end of a series of **`elif`** statements to handle any condition that does not match the conditions specified in the **`elif`** statements. We call the condition handled by the **`else`** statement the *default* condition. If an **`if`/`elif`** structure specifies an **`else`** statement, it must be the last statement in the structure.



Good Programming Practice 3.2

*Provide a default condition in **`if`/`elif`** structures. Conditions not explicitly tested in an **`if`/`elif`** structure without a default condition are ignored. Including a default condition focuses the programmer on the need to process exceptional conditions.*



Software Engineering Observation 3.3

A suite can be placed anywhere in a program that a single statement can be placed.

The **`if`** selection structure can contain several statements in its body (suite), and all these statements must be indented. The following example includes a suite in the **`else`** part of an **`if`/`else`** structure that contains two statements. A suite that contains more than one statement is sometimes called a *compound statement*.

```
if grade >= 60:
    print "Passed."
else:
    print "Failed."
    print "You must take this course again."
```

In this case, if **grade** is less than 60, the program executes both statements in the body of the **else** and prints

```
Failed.
You must take this course again.
```

Notice that both statements of the **else** suite are indented. If the statement

```
print "You must take this course again."
```

was not indented, the statement executes regardless of whether the grade is less than 60 or not. This is an example of a *logic error*.

A programmer can introduce two major types of errors into a program: *syntax errors* and logic errors. A syntax error violates the rules of the programming language. Examples of syntax errors include using a keyword as an identifier or forgetting the colon (**:**) after an **if** statement. The interpreter catches a syntax error and displays an error message.

A logic error causes the program to produce unexpected results and may not be caught by the interpreter. A *fatal logic error* causes a program to fail and terminate prematurely. For fatal errors, Python prints an error message called a *traceback* and exits. A *nonfatal logic error* allows a program to continue executing, but produces incorrect results.

Common Programming Error 3.3

Forgetting to indent all the statements in a suite can lead to syntax or logic errors in a program.

The interactive session in Fig. 3.6 attempts to divide two user-entered values and demonstrates one syntax error and two logic errors. The syntax error is contained in the line

```
print value1 +
```

The **+** operator needs a right-hand operand, so the interpreter indicates a syntax error.

The first logic error is contained in the line

```
print value1 + value2
```

The intention of this line is to **print** the sum of the two user-entered integer values. However, the strings were not converted to integers, thus the statement does not produce the desired result. Instead, the statement produces the concatenation of the two strings—formed by linking the two strings together. Notice that the interpreter does not display any messages because the statement is legal.

The second logic error occurs in the line

```
print int( value1 ) / int( value2 )
```

The program does not check whether the second user-entered value is **0**, so the program attempts to divide by zero. Dividing by zero is a fatal logic error.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> value1 = raw_input( "Enter a number: " )
Enter a number: 3
>>> value2 = raw_input( "Enter a number: " )
Enter a number: 0
>>> print value1 +
  File "<stdin>", line 1
    print value1 +
                 ^
SyntaxError: invalid syntax
>>> print value1 + value2
30
>>> print int( value1 ) / int( value2 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Fig. 3.6 Syntax and logic errors.

Common Programming Error 3.4

An attempt to divide by zero causes a fatal logic error.

Just as multiple statements can be placed anywhere a single statement can be placed, it is possible to have no statements at all, (i.e., *empty statements*). The empty statement is represented by placing keyword **`pass`** where a statement normally resides (Fig. 3.7).

Common Programming Error 3.5

All control structures must contain at least one statement. A control structure that contains no statements causes a syntax error.

3.7 `while` Repetition Structure

A *repetition structure* allows the programmer to specify that a program should repeat an action while some condition remains true. The pseudocode statement

While there are more items on my shopping list
Purchase next item and cross it off my list

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> if 1 < 2:
...     pass
...
```

Fig. 3.7 Keyword `pass`.

describes the repetition that occurs during a shopping trip. The condition, “there are more items on my shopping list” is either true or false. If it is true, the program performs the action “Purchase next item and cross it off my list.” This action is performed repeatedly while the condition remains true.

The statement(s) contained in the *while* repetition structure constitute the body (suite) of the *while*. The *while* structure body can consist of a single statement or multiple statements. Eventually, the condition should evaluate to false (in the above example, when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the program executes the first statement after the repetition structure.

Common Programming Error 3.6

A logic error, called an infinite loop *(the repetition structure never terminates), occurs when an action that causes the condition in the* **`while`** *structure to become false is missing from the body of a* **`while`** *structure.*

Common Programming Error 3.7

Spelling the keyword **`while`** *with an uppercase* **`W`***, as in* **`While`** *(remember that Python is a case-sensitive language), is a syntax error. All of Python’s reserved keywords, such as* **`while`**, **`if`**, **`elif`** *and* **`else`***, contain only lowercase letters.*

As an example of a **`while`** structure, consider a program segment designed to find the first power of 2 larger than 1000. Suppose variable **`product`** has been created and initialized to 2. When the following **`while`** repetition structure finishes executing, product will contain the desired answer:

```
product = 2

while product <= 1000:
   product = 2 * product
```

At the start of the **`while`** structure, **`product`** is 2. The variable **`product`** is multiplied by 2, successively taking on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024. When the value of **`product`** equals 1024, the **`while`** structure condition, **`product <= 1000`**, evaluates to false. This terminates the repetition—the final value of **`product`** is 1024. Program execution continues with the next statement after the **`while`** structure.

The flowchart of Fig. 3.8 illustrates the flow of control in the **`while`** structure that corresponds to the preceding **`while`** structure. Once again, note that (besides small circles and arrows) the flowchart contains a rectangle symbol and a diamond symbol.

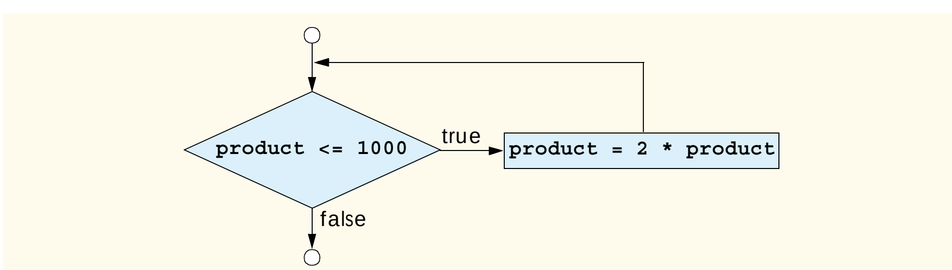


Fig. 3.8 **`while`** repetition structure flowchart.

Imagine a bin of empty **`while`** structures that can be stacked and nested with other control structures to implement an algorithm's flow of control. The empty rectangles and diamonds are then filled in with appropriate actions and decisions. The flowchart shows the repetition. The flowline emerging from the rectangle wraps back to the decision that is tested each time through the loop until the decision becomes false. Then, the **`while`** structure exits and control passes to the next statement in the program.

3.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)

To illustrate how algorithms are developed, we solve several variations of a class-averaging problem. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (integers in the range 0 –100) for this quiz are available. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem requests each of the grades, performs the averaging calculation and prints the result.

Using pseudocode, we list the actions to be executed and specify the order in which these actions should be executed. We use *counter-controlled repetition* to input the grades one at a time. This technique uses a variable called a *counter* to control the number of times a set of statements executes. Repetition terminates when the counter exceeds 10. In this section, we present a pseudocode algorithm (Fig. 3.9) and the corresponding program (Fig. 3.10). In the next section, we show how to develop pseudocode algorithms. Counter-controlled repetition often is called *definite repetition* because the number of repetitions is known before the loop begins executing.

Note the references in the algorithm to the variables *total* and *counter*. In the program of Fig. 3.10, the variable **`total`** (line 5) accumulates the sum of a series of values, while the variable **`counter`** counts—in this case, it counts the number of user-entered grades. Variables that store totals normally are initialized to zero.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 3.9 Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```
1  # Fig. 3.10: fig03_10.py
2  # Class average program with counter-controlled repetition.
3
4  # initialization phase
5  total = 0           # sum of grades
6  gradeCounter = 1    # number of grades entered
7
8  # processing phase
9  while gradeCounter <= 10:                # loop 10 times
10    grade = raw_input( "Enter grade: " ) # get one grade
11    grade = int( grade )    # convert string to an integer
12    total = total + grade
13    gradeCounter = gradeCounter + 1
14
15 # termination phase
16 average = total / 10          # integer division
17 print "Class average is", average
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Fig. 3.10 Counter-controlled repetition used to solve class-average problem.



Good Programming Practice 3.3

Initialize counters and totals.

Lines 5–6 are assignment statements that initialize **total** to 0 and **gradeCounter** to 1. Line 9 indicates that the **while** structure should continue as long as **gradeCounter**'s value is less than or equal to 10.

Lines 10–11 correspond to the pseudocode statement *Input the next grade.* Function **raw_input** displays the prompt "**Enter grade:**" on the screen and accepts user input. Line 11 converts the user-entered string to an integer.

Next, the program updates **total** with the new **grade** entered by the user—line 12 adds **grade** to the previous value of **total** and assigns the result to **total**.

Then, the program increments the variable **gradeCounter** to indicate that a grade has been processed. Line 13 increments **gradeCounter** by one, allowing the condition in the **while** structure to evaluate to false and terminate the loop.

Line 16 executes after the **while** structure terminates and assigns the results of the average calculation to variable **average**. Line 17 displays the string **"Class average is"**, followed by a space (inserted by **print**), followed by the value of variable **average**.

Note that the averaging calculation in the program produces an integer result. Actually, the sum of the grades in this example is 817, which, when divided by 10, yields 81.7—a number with a decimal point. We discuss how to deal with floating-point numbers in the next section.

In Fig. 3.10, if line 16 used **`gradeCounter`** rather than 10 for the calculation, the output for this program would display an incorrect value, **`74`**, because **`gradeCounter`** contains the values **`11`**, after the termination of the **`while`** loop. Fig. 3.11 uses an interactive session to demonstrate the value of **`gradeCounter`** after the **`while`** loop iterates ten times.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)

Let us generalize the class-average problem. Consider the following problem:

> *Develop a class-averaging program that processes an arbitrary number of grades each time the program is executed.*

In the first class-average example, the program knows the number of grades (10) to be entered by the user. In this example, no indication is given of how many grades will be entered. The program processes an arbitrary number of grades. How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a *sentinel value* (also called a *signal value*, a *dummy value* or a *flag value*) to indicate "end of data entry." The user inputs grades until all legitimate grades have been entered. The user then inputs the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition often is called *indefinite repetition* because the number of repetitions is not known before the start of the loop.

Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value. As grades on a quiz normally are nonnegative integers, –1 is an acceptable sentinel value for this problem. Thus, executing the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program then computes and prints the class average for the grades 95, 96, 75, 74 and 89.

Common Programming Error 3.8

Choosing a sentinel value that is a legitimate data value results in a logic error.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> gradeCounter = 1
>>> while gradeCounter <= 10:
...     gradeCounter = gradeCounter + 1
...
>>> print gradeCounter
11
```

Fig. 3.11 Counter value used after termination of counter-controlled loop.

We approach the class-average program with a technique called *top-down, stepwise refinement*, which is essential to the development of well-structured programs. We begin with a pseudocode representation of the *top:*

Determine the class average for the quiz

The top is a single statement that conveys the overall function of the program. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top (as in this case) rarely conveys a sufficient amount of detail from which to write the Python program. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they need to be performed. This results in the following *first refinement*:

Initialize variables
Input, sum and count the quiz grades
Calculate and print the class average

In this case, the sequence control structure is used—the steps listed are executed successively.

Software Engineering Observation 3.4

Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

Software Engineering Observation 3.5

Many programs can be divided logically into three phases: An initialization phase *which initializes the program variables; a* processing phase *which inputs data values and adjusts program variables accordingly; and a* termination phase *which calculates and prints the final results.*

The preceding *Software Engineering Observation* often is all you need for the first refinement in the top-down process. To proceed to the next level of refinement (i.e., the *second refinement*), we commit to specific variables. The program needs to maintain a running total of the numbers, a count of how many numbers have been processed, a variable that contains the value of each grade and a variable that contains the calculated average. The pseudocode statement

Initialize variables

can be refined as follows:

Initialize total to zero
Initialize counter to zero

The pseudocode statement

Input, sum and count the quiz grades

requires a repetition structure (i.e., a loop) that successively inputs each grade. We do not know how many grades will be entered, so we use sentinel-controlled repetition. The user inputs legitimate grades successively. After the last legitimate grade has been entered, the user inputs the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when it has been entered. The second refinement of the preceding pseudocode statement is

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
Add this grade into the running total
Add one to the grade counter
Input the next grade (possibly the sentinel)

The pseudocode statement

Calculate and print the class average

can be refined as follows:

If the counter is not equal to zero
Set the average to the total divided by the counter
Print the average
else
Print "No grades were entered"

Notice that we are testing for the possibility of division by zero—a fatal logic error which, if undetected, causes the program to fail (often called *bombing* or *crashing*). The complete second refinement of the pseudocode for the class average problem is shown in Fig. 3.12.

Good Programming Practice 3.4

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as by printing an error message) rather than allowing the fatal error to occur. In chapter 12, we discuss how to write programs that recognize such errors and take appropriate action. This is known as exception handling.

In Fig. 3.9 and Fig. 3.12, we included some blank lines in the pseudocode to improve the readability of the pseudocode. The blank lines separate these statements into their various phases.

```
Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

Fig. 3.12 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem.

The pseudocode algorithm in Fig. 3.12 solves the more general class-averaging problem. This algorithm was developed after two refinements; sometimes more refinements are necessary.

Software Engineering Observation 3.6

The programmer terminates the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the programmer to convert the pseudocode to Python. After this step, implementing the Python program normally is straightforward.

Figure 3.13 shows the Python program and a sample execution. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point, (i.e., a real number). The integer data type cannot represent real numbers. The program uses the floating-point data type to handle numbers with decimal points and introduces function **float**, which forces the averaging calculation to produce a floating-point numeric result.

```
# Fig. 3.13: fig03_13.py
# Class average program with sentinel-controlled repetition.

# initialization phase
total = 0            # sum of grades
gradeCounter = 0     # number of grades entered

# processing phase
grade = raw_input( "Enter grade, -1 to end: " )    # get one grade
grade = int( grade )    # convert string to an integer

while grade != -1:
   total = total + grade
   gradeCounter = gradeCounter + 1
   grade = raw_input( "Enter grade, -1 to end: " )
   grade = int( grade )

# termination phase
if gradeCounter != 0:
   average = float( total ) / gradeCounter
   print "Class average is", average
else:
   print "No grades were entered"
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.5
```

Fig. 3.13 Sentinel-controlled repetition used to solve class-average problem.

In this example, we see that control structures can be stacked on top of one another (in sequence) just as a child stacks building blocks. The **`while`** structure (lines 12–16) is immediately followed by an **`if`**/**`else`** structure (lines 19–23) in sequence. Much of the code in this program is identical to the code in Fig. 3.10, so in this section, we will concentrate on the new features and issues.

Line 6 initializes the variable **`gradeCounter`** to 0, because no grades have been entered. To keep an accurate record of the number of grades entered, variable **`gradeCounter`** is incremented only when a grade value is entered.

Good Programming Practice 3.5

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user of the sentinel value.

Study the difference between the program logic for sentinel-controlled repetition in Fig. 3.13 and counter-controlled repetition in Fig. 3.10. In counter-controlled repetition, the program reads a value from the user during each pass of the **`while`** structure, for a specified number of passes. In sentinel-controlled repetition, the program reads one value (lines 9–10) before the program reaches the **`while`** structure. This value determines whether the program's flow of control should enter the body of the **`while`** structure. If the **`while`** structure condition is false (i.e., the user has already typed the sentinel), the program does not execute the **`while`** loop (no grades were entered). On the other hand, if the condition is true, the program executes the **`while`** loop and processes the value entered by the user (i.e., adds the **`grade`** to **`total`**). After processing the **`grade`**, the program requests the user to enter another **`grade`**. After executing the last (indented) line of the **`while`** loop (line 16), execution continues with the next test of the **`while`** structure condition, using the new value just entered by the user to determine whether the **`while`** structure's body should execute again. Notice that the program requests the next value before evaluating the **`while`** structure. This allows for determining whether the value just entered by the user is the sentinel value *before* processing the value (i.e., adding it to **`total`**). If the value entered is the sentinel value, the **`while`** structure terminates, and the value is not added to **`total`**.

Lines 9–10 and 15–16 contain identical lines of code. In Section 3.15, we introduce programming constructs that help the programmer avoid repeating code.

Averages do not always evaluate to integer values. Often, an average is a value that contains a fractional part, such as 7.2 or –93.5. These values are referred to as *floating-point numbers*.

The calculation **`total / gradeCounter`** results in an integer, because **`total`** and **`counter`** contain integer values. Dividing two integers results in integer division, in which any fractional part of the calculation is discarded (i.e., truncated). The calculation is performed first, the fractional part is discarded before assigning the result to **`average`**. To produce a floating-point calculation with integer values, convert one (or both) of the values to a floating-point value with function **`float`**. Recall that functions are pieces of code that accomplish a task; in line 20, function **`float`** converts the integer value of variable sum to a floating-point value. The calculation now consists of a floating-point value divided by the integer **`gradeCounter`**.

The Python interpreter knows how to evaluate expressions in which the data types of the operands are identical. To ensure that the operands are of the same type, the interpreter

performs an operation called *promotion* (also called *implicit conversion*) on selected operands. For example, in an expression containing integer and floating-point data, integer operands are *promoted* to floating point. In our example, the value of **`gradeCounter`** is promoted to a floating-point number. Then, the calculation is performed, and the result of the floating-point division is assigned to variable **`average`**.

Common Programming Error 3.9

Assuming that all floating-point numbers are precise can lead to incorrect results. Most computers approximate floating-point numbers.

Despite the fact that floating-point numbers are not precise, they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that calling this number simply 98.6 is adequate for most applications.

Another way floating-point numbers develop is through division. When we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates a fixed amount of space to hold such a value, so the stored floating-point value only can be an approximation.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)

Let us work another complete problem. We once again formulate the algorithm using pseudocode and top-down, stepwise refinement and we develop a corresponding Python program. Consider the following problem statement:

> *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, several of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.*
>
> *Your program should analyze the results of the exam as follows:*
>
> 1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.*
> 2. *Count the number of test results of each type.*
> 3. *Display a summary of the test results indicating the number of students who passed and the number of students who failed.*
> 4. *If more than 8 students passed the exam, print the message "Raise tuition."*

After reading the problem statement carefully, we make the following observations about the problem:

1. The program must process 10 test results. A counter-controlled loop will be used.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it is a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)

3. Two counters are used—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
4. After the program has processed all the results, it must decide if more than eight students passed the exam.

Let us proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

Analyze exam results and decide if tuition should be raised

Once again, it is important to emphasize that the top is a complete representation of the program, but several refinements are likely to be needed before the pseudocode can evolve naturally into a Python program. Our first refinement is

Initialize variables
Input the ten exam grades and count passes and failures
Print a summary of the exam results and decide if tuition should be raised

Here, too, even though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. We need counters to record the passes and failures, a counter to control the looping process and a variable to store the user input. The pseudocode statement

Initialize variables

can be refined as follows:

Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

Notice that only the counters for the number of passes, number of failures and number of students are initialized. The pseudocode statement

Input the ten exam grades and count passes and failures

requires a loop that successively inputs the result of each exam. Here it is known in advance that there are precisely ten exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., *nested* within the loop), a double-selection structure determines whether each exam result is a pass or a failure and increments the appropriate counter accordingly. The refinement of the preceding pseudocode statement is

```
While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    else
        Add one to failures

    Add one to student counter
```

Notice the use of blank lines to set off the *If/else* control structure to improve program readability. The pseudocode statement

Print a summary of the exam results and decide if tuition should be raised

may be refined as follows:

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Raise tuition"

The complete second refinement appears in Fig. 3.14. Notice that the pseudocode also uses blank lines to set off the while structure for program readability.

This pseudocode is now sufficiently refined for conversion to Python. Figure 3.15 shows the Python program and two sample executions.

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Raise tuition"
```

Fig. 3.14 Pseudocode for examination-results problem.

```python
# Fig. 3.15: fig03_15.py
# Analysis of examination results.

# initialize variables
passes = 0                    # number of passes
failures = 0                  # number of failures
studentCounter = 1            # student counter

# process 10 students; counter-controlled loop
while studentCounter <= 10:
   result = raw_input( "Enter result (1=pass,2=fail): " )
   result = int( result )   # one exam result
```

Fig. 3.15 Examination-results problem. (Part 1 of 2.)

```
13
14    if result == 1:
15       passes = passes + 1
16    else:
17       failures = failures + 1
18
19    studentCounter = studentCounter + 1
20
21 # termination phase
22 print "Passed", passes
23 print "Failed", failures
24
25 if passes > 8:
26    print "Raise tuition"
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4
```

Fig. 3.15 Examination-results problem. (Part 2 of 2.)

Note that line 14 uses the equality operator (`==`) to test whether the value of variable **`result`** equals 1. Be careful not to confuse the equality operator with the assignment symbol (`=`). Such confusion can cause syntax or logic errors in Python.

Common Programming Error 3.10

Using the `=` *symbol for equality in a conditional statement is a syntax error.*

Common Programming Error 3.11

Using operator `==` *for assignment is a logic error.*

Software Engineering Observation 3.7

Experience has shown that the most difficult part of solving a problem on a computer is developing an algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working Python program from the algorithm normally is straightforward.

Software Engineering Observation 3.8

Many experienced programmers write programs without ever using program-development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this may work for simple and familiar problems, it can lead to serious errors and delays on large, complex projects.

3.11 Augmented Assignment Symbols

Python provides several *augmented assignment symbols* for abbreviating assignment expressions. For example, the statement

```
c = c + 3
```

can be abbreviated with the *augmented addition assignment symbol* `+=` as

```
c += 3
```

The `+=` symbol adds the value of the expression on the right of the `+=` sign to the value of the variable on the left of the sign and stores the result in the variable on the left of the sign. Any statement of the form

variable `=` *variable operator expression*

where *operator* is a binary operator, such as `+`, `-`, `**`, `*`, `/`, or `%`, can be written in the form

variable operator`=` *expression*

A statement that uses an augmented assignment symbol is called an *augmented assignment statement*. Figure 3.16 shows the augmented arithmetic assignment symbols.

Assignment symbol	Sample expression	Explanation	Assigns
Assume: `c = 3, d = 5, e = 4, f = 2, g = 6, h = 12`			
`+=`	`c += 7`	`c = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`

Fig. 3.16 Augmented arithmetic assignment symbols. (Part 1 of 2.)

Assignment symbol	Sample expression	Explanation	Assigns
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`**=`	`f **= 3`	`f = f ** 3`	`8` to `f`
`/=`	`g /= 3`	`g = g / 3`	`2` to `g`
`%=`	`h %= 9`	`h = h % 9`	`3` to `h`

Fig. 3.16 Augmented arithmetic assignment symbols. (Part 2 of 2.)

Portability Tip 3.1

Augmented assignment symbols were introduced in Python version 2.0. Attempting to use an augmented assignment symbol with an earlier version of Python is a syntax error.

Common Programming Error 3.12

Attempting to use an augmented assignment before the variable to the left of the assignment symbol has been initialized is an error.

3.12 Essentials of Counter-Controlled Repetition

Counter-controlled repetition requires the following:

1. the *name* of a control variable (or loop counter),
2. the *initial value* of the control variable,
3. the amount of *increment* (or *decrement*) by which the control variable is modified each time through the loop (also known as each iteration of the loop), and
4. the condition that tests for the *final value* of the control variable (i.e., whether looping should continue).

Consider the simple program in Fig. 3.17, which prints the numbers from 0 to 9. Line 4 names the control variable (**`counter`**) and sets it to an *initial value* of **`0`**. Line 8 in the **`while`** structure *increments* the loop counter by 1 for each iteration of the loop. The loop-continuation condition in the **`while`** structure tests for whether the value of the control variable is less than **`10`**. The loop terminates when the control variable is greater than or equal to **`10`** (i.e., **`counter`** becomes **`10`**).

```
1  # Fig. 3.17: fig03_17.py
2  # Counter-controlled repetition.
3
4  counter = 0
5
6  while counter < 10:
7     print counter
8     counter += 1
```

Fig. 3.17 Counter-controlled repetition. (Part 1 of 2.)

```
0
1
2
3
4
5
6
7
8
9
```

Fig. 3.17 Counter-controlled repetition. (Part 2 of 2.)

Common Programming Error 3.13

Because floating-point values may be approximate, controlling the counting of loops with floating-point variables may result in imprecise counter values and inaccurate tests for termination. Programs should control counting loops with integer values.

Good Programming Practice 3.6

Put a blank line before and after each control structure to make it stand out in the program.

Good Programming Practice 3.7

Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of indentation.

Good Programming Practice 3.8

Inserting a blank line above and below each control structure, and indenting the body of each control structure, give programs a two-dimensional appearance that enhances readability.

3.13 `for` Repetition Structure

The **`for`** repetition structure handles all the details of counter-controlled repetition. To illustrate the power of **`for`**, let us rewrite the program of Fig. 3.17. Figure 3.18 shows the result.

The program operates as follows. When the **`for`** structure begins executing, function ***`range`*** creates a *sequence* of values in the range 0–9 (Fig. 3.19). The first value in this sequence is assigned to variable **`counter`**, and the body of the **`for`** structure (line 6) executes. For each subsequent value in the sequence, the value is assigned to variable **`counter`**, and the body of the **`for`** structure executes. This process continues until all values in the sequence have been processed.

Fig. 3.19 shows the sequence returned by function **`range`**. This sequence is a Python *list* containing integers in the range 0–9. Note that values in a list are enclosed in square brackets (e.g., **`[]`**) and separated by commas. Lists are covered in detail in Chapter 5, Lists, Tuples and Dictionaries.

Notice that the last value of the sequence returned by function **`range`** is one less than the argument passed to the function. If the programmer incorrectly wrote

```
for counter in range( 9 ):
   print counter
```

then the loop executes nine times. This is a common logic error called an *off-by-one* error.

```
# Fig. 3.18: fig03_18.py
# Counter-controlled repetition with the
# for structure and range function.

for counter in range( 10 ):
   print counter
```

```
0
1
2
3
4
5
6
7
8
9
```

Fig. 3.18 Counter-controlled repetition with the **for** structure.

Function **range** can take one, two or three arguments. If we pass one argument to the function (as in Fig. 3.19), that argument, called ***end***, is one greater than the *upper bound* (highest value) of the sequence. In this case, **range** returns a sequence in the range:

```
0–( end-1 )
```

If we pass two arguments, the first argument, called ***start***, is the *lower bound*—the lowest value in the returned sequence—and the second argument is **end**. In this case, **range** returns a sequence in the range:

```
( start )–( end-1 )
```

If we pass three arguments, the first two arguments are **start** and **end**, respectively, and the third argument, called ***increment***, is the *increment value*. The sequence produced by a call to **range** with an increment value progresses from **start** to **end** in multiples of the increment value. If **increment** is positive, the last value in the sequence is the largest multiple less than **end**. The following three calls to **range** produce the same sequence as in Fig. 3.19.

```
range( 10 )
range( 0, 10 )
range( 0, 10, 1 )
```

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> range( 10 )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Fig. 3.19 Function **range**.

Common Programming Error 3.14

*Forgetting that the first value of the sequence returned by function **`range`**, if no lower bound is provided, is zero can lead to an off-by-one logic error.*

Common Programming Error 3.15

*Forgetting that the last value of the sequence returned by function **`range`** is one less than the value of the function's **`end`** argument can lead to an off-by-one logic error.*

The increment value of **`range`** also can be negative. In this case, it is a decrement and the sequence produced progresses downwards from **`start`** to **`end`** in multiples of the increment value. The last value in the sequence is the smallest multiple greater than **`end`** (Fig. 3.20).

The sequence used in a **`for`** structure does not have to be generated using the **`range`** function. The general format of the **`for`** structure is

```
for element in sequence:
    statement(s)
```

where *sequence* is a set of items (sequences are explained in detail in Chapter 5). At the first iteration of the loop, variable *element* is assigned the first item in the sequence and *statement* is executed. At each subsequent iteration of the loop, variable *element* is assigned the next item in the sequence before the execution of *statement*. Once the loop has been executed once for each item in the *sequence*, the loop terminates. In most cases, the **`for`** structure can be represented by an equivalent **`while`** structure, as in

```
initialization

while loopContinuationTest:
    statement(s)
    increment
```

where the *initialization* expression initializes the loop's control variable, *loopContinuationTest* is the loop-continuation condition and *increment* increments the control variable.

Common Programming Error 3.16

*Creating a **`for`** structure that contains no body statements is a syntax error.*

If the *sequence* part of the **`for`** structure is empty (i.e., the sequence contains no values), the program does not perform the body of the **`for`** structure. Instead, execution proceeds with the statement following the **`for`** structure.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> range( 10, 0, -1 )
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Fig. 3.20 Function **`range`** with a third value.

Programs frequently display the control variable (*element*) or use it in calculations in the loop body. However, this use is not required. It is common to use the control variable for controlling repetition while never mentioning it in the body of the **for** structure.

Good Programming Practice 3.9

Avoid changing the value of the control variable in the body of a ***for*** *loop, because this practice can cause subtle logic errors.*

The flowchart of the **for** structure is similar to that of the **while** structure. Figure 3.21 illustrates the flowchart of the following **for** statement

```
for x in y:
   print x
```

The flowchart shows the initialization and the update processes. Note that update occurs each time *after* the program performs the body statement. Besides small circles and arrows, the flowchart contains only rectangle symbols and a diamond symbol. The programmer fills the rectangles and diamonds with actions and decisions appropriate to the algorithm.

3.14 Using the for Repetition Structure

The following examples show techniques for varying the control variable (loop counter) in a **for** structure. In each case, we write the appropriate **for** header. Note the change in the third argument to **range** for loops that decrement the control variable.

a) Vary the control variable from **1** to **100** in increments of **1**.

```
for counter in range( 1, 101 ):
```

b) Vary the control variable from **100** to **1** in increments of **-1** (decrements of **1**).

```
for counter in range( 100, 0, -1 ):
```

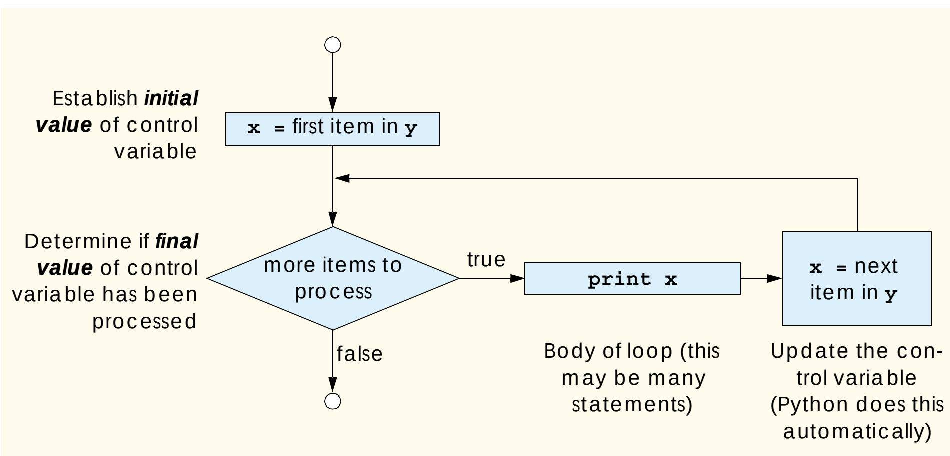


Fig. 3.21 **for** repetition structure flowchart.

c) Vary the control variable from `7` to `77` in steps of `7`.

```
for counter in range( 7, 78, 7 ):
```

d) Vary the control variable from `20` to `2` in steps of `-2`.

```
for counter in range( 20, 1, -2 ):
```

e) Vary the control variable over the following sequence of values: `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for counter in range( 2, 21, 3 ):
```

f) Vary the control variable over the following sequence of values: `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for counter in range( 99, -1, -11 ):
```

The next two examples provide simple applications of the **`for`** structure. The program in Fig. 3.22 uses the **`for`** structure to sum all the even integers from `2` to `100`.

The next example computes compound interest using the **`for`** structure. Consider the following problem statement:

A person invests $1000 in a savings account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

$$a = p (1 + r)^n$$

where

p is the original amount invested (i.e., the principal),
r is the annual interest rate,
n is the number of years and
a is the amount on deposit at the end of the nth year.

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. Figure 3.23 shows the solution. The **`for`** structure executes the body of the loop 10 times, incrementing a control variable (**`year`**) from 1 to 10. In this example, the algebraic expression $(1 + r)^n$ is written as **`( 1 + rate ) ** year`**, where variable **`rate`** represents r and variable **`year`** represents n.

```
# Fig. 3.22: fig03_22.py
# Summation with for.

sum = 0

for number in range( 2, 101, 2 ):
   sum += number

print "Sum is", sum
```

```
Sum is 2550
```

Fig. 3.22 Summation with **`for`**.

```
1  # Fig. 3.23: fig03_23.py
2  # Calculating compound interest.
3
4  principal = 1000.0    # starting principal
5  rate = .05            # interest rate
6
7  print "Year %21s" % "Amount on deposit"
8
9  for year in range( 1, 11 ):
10    amount = principal * ( 1.0 + rate ) ** year
11    print "%4d%21.2f" % ( year, amount )
```

```
Year     Amount on deposit
   1               1050.00
   2               1102.50
   3               1157.63
   4               1215.51
   5               1276.28
   6               1340.10
   7               1407.10
   8               1477.46
   9               1551.33
  10               1628.89
```

Fig. 3.23 `for` structure used to calculate compound interest.

The output statement before the **`for`** loop (line 7) and the output statement in the **`for`** loop (line 11) combine to print the values of the variables **`year`** and **`amount`** with the formatting specified by the **`%`** formatting operator specifications. The characters **`%4d`** specify that the **`year`** column is printed with a field width of four (i.e., the value is printed with at least four character positions). If the value to be output is fewer than four character positions wide, the value is right justified in the field by default. If the value to be output is more than four character positions wide, the field width is extended to accommodate the entire value.

The characters **`%21.2f`** indicate that variable **`amount`** is printed as a float-point value (specified with the character **`f`**) with a decimal point. The column has a total field width of 21 character positions and two digits of precision to the right of the decimal point; the total field width includes the decimal point and the two digits to its right, hence 18 of the 21 positions appear to the left of the decimal point.

Notice that the variables **`amount`**, **`principal`** and **`rate`** are floating point values. We did this for simplicity, because we are dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, this can cause trouble. Here is an example of what can go wrong when using floating point values to represent dollar amounts (assuming that dollar amounts are displayed with two digits to the right of the decimal point): Two dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your printout could appear as

```
  14.23
+ 18.67
-------
  32.91
```

but a person adding the individual numbers as printed would expect the sum to be 32.90. You have been warned!

Good Programming Practice 3.10

Be careful when using floating-point values to perform monetary calculations. Rounding errors may lead to undesired results.

Note that the body of the **for** structure contains the calculation **1.0 + rate** (line 10). In fact, this calculation produces the same result each time through the loop, so repeating the calculation is wasteful. A better solution would be to define a variable (e.g., **finalRate** that references the value of **1.0 + rate** before the start of the **for** structure. Then, replace the calculation **1.0 + rate** (line 10) with variable **finalRate**.

Performance Tip 3.3

Avoid placing expressions whose values do not change inside loops.

3.15 break and continue Statements

Python offers the ***break*** and ***continue*** statements, which alter the flow of control. The **break** statement, when executed in a **while** or **for** structure, causes immediate exit from that structure. Program execution continues with the first statement after the structure. Figure 3.24 demonstrates the **break** statement in a **for** repetition structure. When the **if** structure detects that **x** equals **5**, it executes the **break** statement. This terminates the **for** statement and the program continues with the **print** statement (line 11). The loop outputs four numbers.

Figure 3.25 is a modified version of Fig. 3.13, the class-average program illustrating sentinel-controlled repetition. This version eliminates the repeated code found in the original program. Line 9 introduces an infinite **while** loop. The condition of the **while** loop never evaluates to false because 1 is always true. Lines 10–11 prompt the user for a grade and convert the input to an integer. If the grade is the sentinel value, –1, the program exits the loop (line 16).

```
1  # Fig. 3.24: fig03_24.py
2  # Using the break statement in a for structure.
3
4  for x in range( 1, 11 ):
5
6     if x == 5:
7        break
8
9     print x,
10
11 print "\nBroke out of loop at x =", x
```

Fig. 3.24 **break** statement used in a **for** structure. (Part 1 of 2.)

```
1 2 3 4
Broke out of loop at x = 5
```

Fig. 3.24 **`break`** statement used in a **`for`** structure. (Part 2 of 2.)

```
# Fig. 3.25: fig03_25.py
# Using the break statement to avoid repeating code
# in the class-average program.

# initialization phase
total = 0           # sum of grades
gradeCounter = 0    # number of grades entered

while 1:
   grade = raw_input( "Enter grade, -1 to end: " )
   grade = int( grade )

   # exit loop if user inputs -1
   if grade == -1:
      break

   total += grade
   gradeCounter += 1

# termination phase
if gradeCounter != 0:
   average = float( total ) / gradeCounter
   print "Class average is", average
else:
   print "No grades were entered"
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.5
```

Fig. 3.25 **`break`** statement used to eliminate code repetition.

The ***`continue`*** statement, when executed in a **`while`** or a **`for`** structure, skips the remaining statements in the body of that structure and proceeds with the next iteration of the loop. In **`while`** structures, the loop-continuation test is evaluated immediately after the execution of the **`continue`** statement. In the **`for`** structure, the control variable is assigned the next value in the sequence (if the sequence contains more values). Earlier, we stated that the **`while`** structure usually can represent the **`for`** structure. The one exception occurs when the increment expression in the **`while`** structure follows the **`continue`**

statement. In this case, the increment is not executed before the repetition-continuation condition is tested, and the **while** does not execute in the same manner as the **for**. Figure 3.26 uses the **continue** statement in a **for** structure to skip the output statement in the structure and begin the next iteration of the loop.

Good Programming Practice 3.11

*Some programmers feel that **break** and **continue** violate structured programming. Because the effects of these statements can be achieved by structured programming techniques we discuss, these programmers do not use **break** and **continue**.*

3.16 Logical Operators

So far, we have studied *simple conditions*, such as **counter <= 10**, **total > 1000** and **number != sentinelValue**. We have expressed these conditions in terms of the relational operators **>**, **<**, **>=** and **<=** and the equality operators **==** and **!=**. Each decision tested precisely one condition. To test multiple conditions while making a decision, we performed these tests in separate statements or in nested **if** or **if/else** structures.

Python provides *logical operators* that are used to form more complex conditions by combining simple conditions. The logical operators are ***and*** *(logical AND),* ***or*** *(logical OR) and* ***not*** *(logical NOT*, also called *logical negation).* We now consider examples of each of these operators.

Suppose we wish to ensure that two conditions are *both* true before we choose a certain path of execution. In this case, we can use the logical **and** operator as follows:

```
if gender == "Female" and age >= 65:
   seniorFemales += 1
```

This **if** statement contains two simple conditions. The condition **gender == "Female"** is evaluated here to determine whether a person is a female. The condition **age >= 65** is evaluated to determine whether a person is a senior citizen. The simple condition to the left of the **and** operator is evaluated first, because the precedence of **==** is higher than the precedence of **and**. If necessary, the simple condition to the right of the **and** operator is evaluated next, because the precedence of **>=** is higher than the precedence of **and** (as we will discuss shortly, the right side of a logical AND expression is evaluated only if the left side is true). The **if** statement then considers the combined condition:

```
# Fig. 3.26: fig03_26.py
# Using the continue statement in a for/in structure.

for x in range( 1, 11 ):

   if x == 5:
      continue

   print x,

print "\nUsed continue to skip printing the value 5"
```

Fig. 3.26 **continue** statement used in a **for** structure. (Part 1 of 2.)

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 3.26 **continue** statement used in a **for** structure. (Part 2 of 2.)

```
gender == "Female" and age >= 65
```

This condition is true only if both of the simple conditions are true. Finally, if this combined condition is indeed true, then the count of **seniorFemales** is incremented by 1. If either or both of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the **if**. The preceding combined condition can be made more readable by adding redundant parentheses

```
( gender == "Female" ) and ( age >= 65 )
```

The table of Fig. 3.27 summarizes the **and** operator. The table shows all four possible combinations of false and true values for **expression1** and **expression2**. Such tables are often called *truth tables*.

Python evaluates to false or true all expressions that include relational operators and equality operators. A simple condition (e.g., **age >= 65**) that is false evaluates to the integer value 0; a simple condition that is true evaluates to the integer value 1. A Python expression that evaluates to the value 0 is false; a Python expression that evaluates to a non-zero integer value is true. The interactive session of Fig. 3.28 demonstrates these concepts.

Lines 5–10 of the interactive session demonstrate that the value 0 is false. Lines 11–18 show that any non-zero integer value is true. The simple condition in line 19 evaluates to true (line 20). The combined conditions in lines 21 and 23 demonstrate the return values of the **and** operator. If a combined condition evaluates to false (line 21), the **and** operator returns the first value which evaluated to false (line 22). Conversely, if the combined condition evaluates to true (line 23), the **and** operator returns the last value in the condition (line 24).

Now let us consider the **or** (logical OR) operator. Suppose we wish to ensure at some point in a program that either one *or* both of two conditions are true before we choose a certain path of execution. In this case, we use the **or** operator, as in the following program segment:

```
if semesterAverage >= 90 or finalExam >= 90:
   print "Student grade is A"
```

expression1	expression2	expression1 and expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 3.27 Truth table for the **and** (logical AND) operator.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> if 0:
...     print "0 is true"
... else:
...     print "0 is false"
...
0 is false
>>> if 1:
...     print "non-zero is true"
...
non-zero is true
>>> if -1:
...     print "non-zero is true"
...
non-zero is true
>>> print 2 < 3
1
>>> print 0 and 1
0
>>> print 1 and 3
3
```

Fig. 3.28 Truth values.

This preceding condition also contains two simple conditions. The simple condition **semesterAverage >= 90** is evaluated to determine whether the student deserves an "A" in the course because of a solid performance throughout the semester. The simple condition **finalExam >= 90** is evaluated to determine whether the student deserves an "A" in the course because of an outstanding performance on the final exam. The **if** statement then considers the combined condition

```
semesterAverage >= 90 or finalExam >= 90
```

and awards the student an "A" if either one or both of the simple conditions are true. Note that the message **Student grade is A** is not printed when both of the simple conditions are false. Fig. 3.29 is a truth table for the logical OR operator (**or**).

expression1	expression2	expression1 **or** expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 3.29 Truth table for the **or** (logical OR) operator.

If a combined condition evaluates to true, the **or** operator returns the first value which evaluated to true. Conversely, if the combined condition evaluates to false, the **or** operator returns the last value in the condition.

The **and** operator has a higher precedence than the **or** operator. Both operators associate from left to right. An expression containing **and** or **or** operators is evaluated until its truth or falsity is known. This is called *short circuit evaluation*. Thus, evaluation of the expression

```
gender == "Female" and age >= 65
```

will stop immediately if **gender** is not equal to **"Female"** (i.e., the entire expression is false), but continue if **gender** is equal to **"Female"** (i.e., the entire expression could still be true, if the condition **age >= 65** is true).

Performance Tip 3.4

In expressions using operator ***and****, if the separate conditions are independent of one another, make the condition that is more likely to be* false *the left-most condition. In expressions using operator* ***or****, make the condition that is more likely to be* true *the left-most condition. This approach can reduce a program's execution time.*

Python provides the **not** (logical negation) operator to enable a programmer to "reverse" the meaning of a condition. Unlike the **and** and **or** operators, which combine two conditions (binary operators), the logical negation operator has a single condition as an operand (i.e., **not** is a *unary* operator). The logical negation operator is placed before a condition when we are interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
if not grade == sentinelValue:
   print "The next grade is", grade
```

Figure 3.30 is a truth table for the logical negation operator. In many cases, the programmer can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the preceding statement can also be written as follows:

```
if grade != sentinelValue:
   print "The next grade is", grade
```

This flexibility can often help a programmer express a condition in a more "natural" or convenient manner.

expression	**not** expression
false	true
true	false

Fig. 3.30 Truth table for operator **not** (logical negation).

Figure 3.31 shows the precedence and associativity of the Python operators introduced to this point. The operators are shown from top to bottom, in decreasing order of precedence.

3.17 Structured-Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design their programs. The field of computer programming is younger than architecture, and our collective wisdom is considerably sparser. We have learned that structured programming produces programs that are easier than unstructured programs to understand and hence are easier to test, debug, modify, and even prove correct in a mathematical sense.

Figure 3.32 summarizes Python's control structures. Small circles are used in the figure to indicate the single entry point and the single exit point of each structure. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control structures and to build structured programs by properly combining control structures in only two simple ways.

For simplicity, single-entry/single-exit control structures are used—there is one way to enter and one way to exit each control structure. Connecting control structures in sequence to form structured programs is simple—the exit point of one control structure is connected to the entry point of the next control structure, so that control structures are simply placed one after another in a program; we have called this "control-structure stacking." The rules for forming structured programs also allow for control structures to be nested.

Figure 3.33 shows the rules for forming properly structured programs. The rules assume that the rectangle flowchart symbol may be used to indicate any action, including input and output. The rules also assume that we begin with the simplest flowchart (Fig. 3.34).

Operators	Associativity	Type
`()`	left to right	parentheses
`**`	right to left	exponentiation
`* / %`	left to right	multiplicative
`+`	left to right	additive
`< <= > >=`	left to right	relational
`== != <>`	left to right	equality
`and`	left to right	logical AND
`or`	left to right	logical OR
`not`	right to left	logical NOT

Fig. 3.31 Operator precedence and associativity.

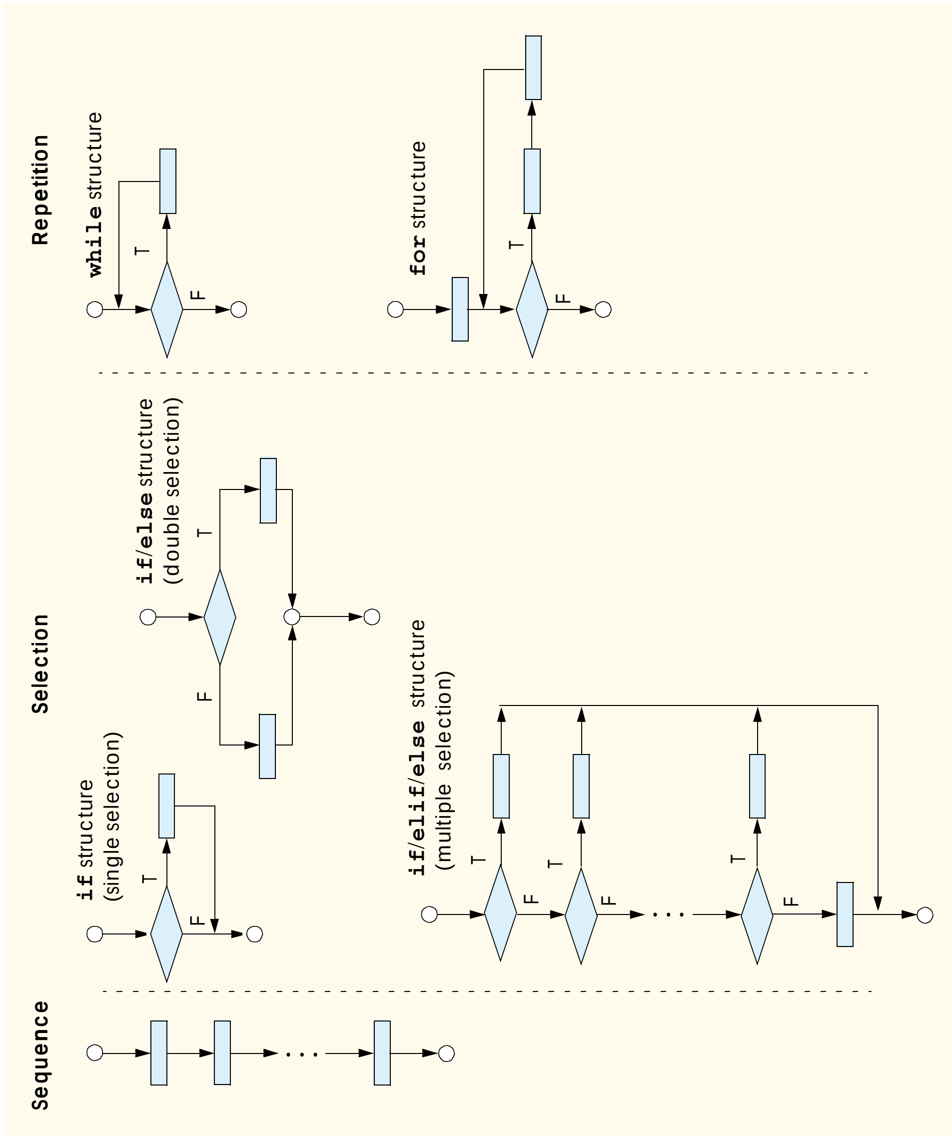


Fig. 3.32 Single-entry/single-exit sequence, selection and repetition structures.

Rules for Forming Structured Programs
1) Begin with the so called simplest flowchart (Fig. 3.34).
2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.

Fig. 3.33 Rules for forming structured programs. (Part 1 of 2.)

Rules for Forming Structured Programs

3) Any rectangle (action) can be replaced by any control structure (sequence, **`if`**, **`if`**/**`else`**, **`if`**/**`elif`**/**`else`**, **`while`** or **`for`**).

4) Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 3.33 Rules for forming structured programs. (Part 2 of 2.)

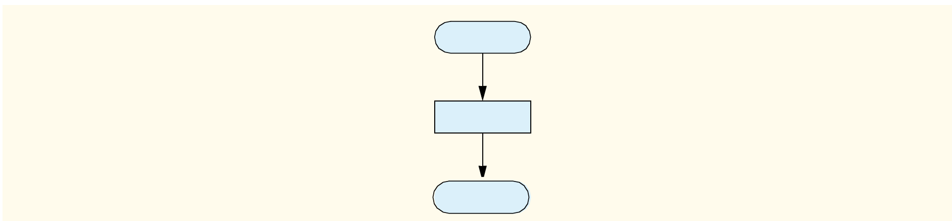


Fig. 3.34 Simplest flowchart.

Applying the rules of Fig. 3.33 always results in a structured flowchart with a neat, building-block appearance. For example, repeatedly applying rule 2 to the simplest flowchart results in a structured flowchart containing many rectangles in sequence (Fig. 3.35). Notice that rule 2 generates a stack of control structures, so let us call rule 2 the *stacking rule*.

Rule 3 is called the *nesting rule*. Repeatedly applying rule 3 to the simplest flowchart results in a flowchart with neatly nested control structures. For example, in Fig. 3.36, the rectangle in the simplest flowchart is first replaced with a double-selection (**`if`**/**`else`**) structure. Then rule 3 is applied again to both of the rectangles in the double-selection structure, replacing each of these rectangles with double-selection structures. The dashed boxes around each of the double-selection structures represent the rectangles that were replaced.

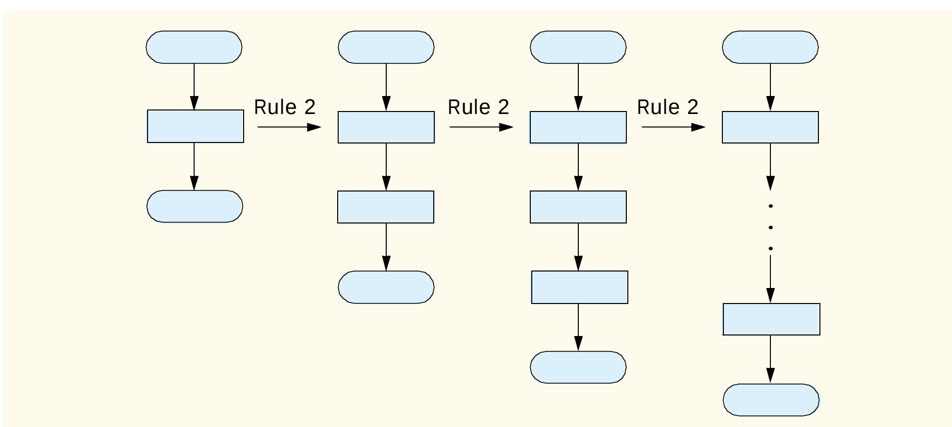


Fig. 3.35 Applying (repeatedly) rule 2 of Fig. 3.33 to the simplest flowchart.

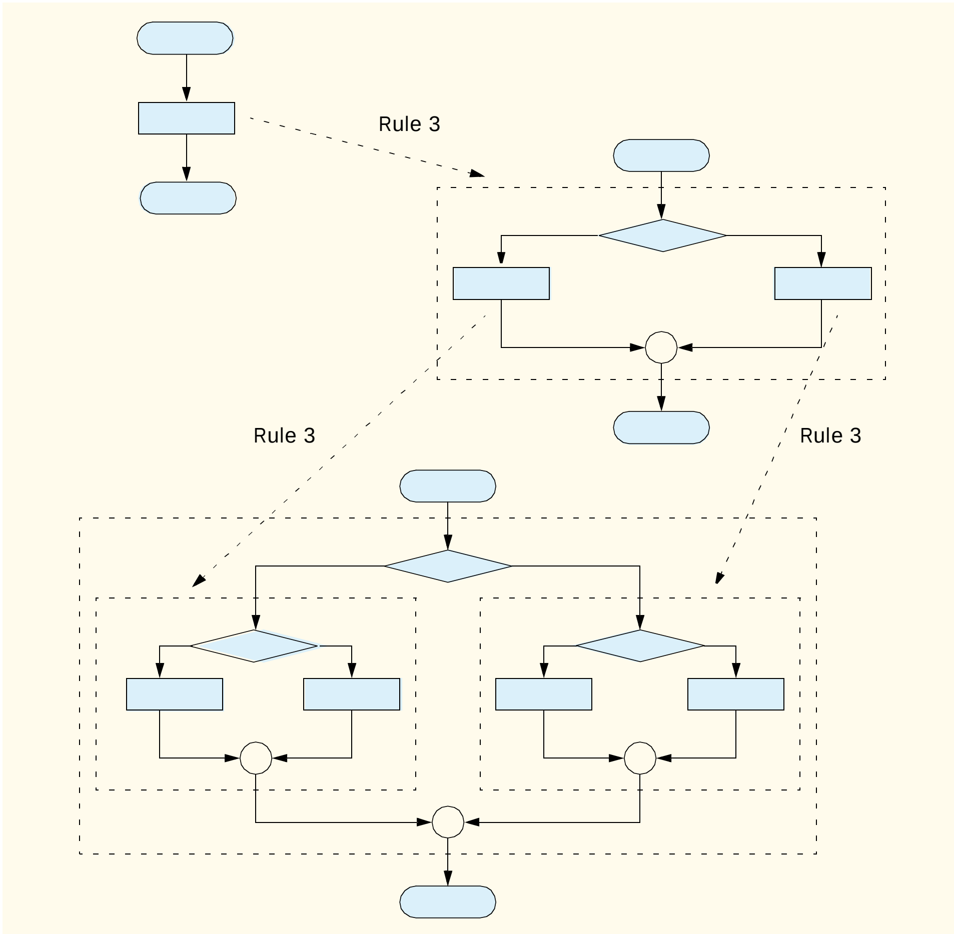


Fig. 3.36 Applying rule 3 of Fig. 3.35 to the simplest flowchart.

Rule 4 generates larger, more involved and more deeply nested structures. The flowcharts that emerge from applying the rules in Fig. 3.33 constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.

The beauty of the structured approach is that we use only six simple single-entry/single-exit pieces, and we assemble them in only two simple ways. Figure 3.37 shows the kinds of stacked building blocks that emerge from applying rule 2 and the kinds of nested building blocks that emerge from applying rule 3. The figure also shows the kind of overlapped building blocks that cannot appear in structured flowcharts (because of the elimination of the `goto` statement).

If the rules in Fig. 3.33 are followed, an unstructured flowchart (such as that in Fig. 3.38) cannot be created. If you are uncertain of whether a particular flowchart is structured, apply the rules of Fig. 3.33 in reverse to try to reduce the flowchart to the simplest flowchart. If the flowchart is reducible to the simplest flowchart, the original flowchart is structured; otherwise, it is not.

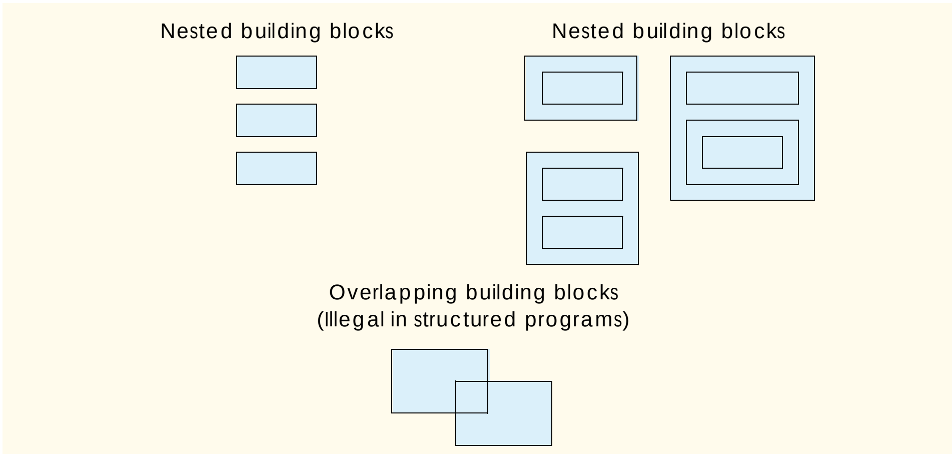


Fig. 3.37 Stacked, nested and overlapped building blocks.

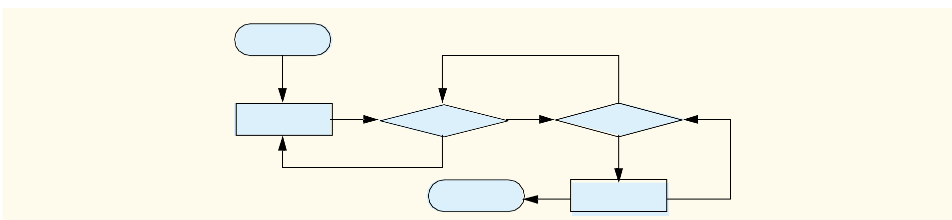


Fig. 3.38 Unstructured flowchart.

Structured programming promotes simplicity. Bohm and Jacopini have given us the result that only three forms of control are needed:

- Sequence
- Selection
- Repetition

Sequence is trivial. Selection is implemented in one of three ways:

- **`if`** structure (single selection)
- **`if`**/**`else`** structure (double selection)
- **`if`**/**`elif`**/**`else`** structure (multiple selection)

In fact, it is straightforward to prove that the simple **`if`** structure is sufficient to provide any form of selection—everything that can be done with the **`if`**/**`else`** structure and the **`if`**/**`elif`**/**`else`** structure can be implemented by combining **`if`** structures (although perhaps not as clearly and efficiently).

Repetition is implemented in one of two ways:

- **`while`** structure
- **`for`** structure

It is straightforward to prove that the **`while`** structure is sufficient to provide any form of repetition. Everything that can be done with the **`for`** structure can be done with the **`while`** structure (although perhaps not as smoothly).

Combining these results illustrates that any form of control ever needed in a Python program can be expressed in terms of the following:

- sequence
- **`if`** structure (selection)
- **`while`** structure (repetition)

Also, these control structures can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

In this chapter, we discussed how to compose programs from control structures containing actions and decisions. In Chapter 4, Functions, we introduce another program-structuring unit, called the *function*. We learn to compose large programs by combining functions that, in turn, are composed of control structures. We also discuss how functions promote software reusability. In Chapter 7, Object-Oriented Programming, we introduce Python's other program-structuring unit, called the *class*. We then create objects from classes and proceed with our treatment of *object-oriented programming (OOP)*.

SUMMARY

- Any computing problem can be solved by executing a series of actions in a specified order. An algorithm solves problems in terms of the actions to be executed and the order in which these actions are executed.
- Specifying the order in which statements execute in a computer program is called program control.
- Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode is similar to everyday English; it is convenient and user-friendly, although it is not an actual computer programming language.
- A carefully prepared pseudocode program can be converted easily to a corresponding Python program. In many cases, this is done simply by replacing pseudocode statements with their Python equivalents.
- Normally, statements in a program execute successively in the order in which they appear. This is called sequential execution. Various Python statements enable the programmer to specify that the next statement to be executed may be other than the next one in sequence. This is called transfer of control.
- The **`goto`** statement allows a programmer to specify a transfer of control to one of a wide range of possible destinations in a program.
- The research of Bohm and Jacopini demonstrated that programs could be written without any **`goto`** statements. The challenge of the era became for programmers to shift their styles to "**`goto`**-less programming."
- Bohm and Jacopini demonstrated that all programs could be written in terms of only three control structures—the sequence, selection and repetition structures.
- The sequence structure is built into Python. Unless directed otherwise, the computer executes Python statements sequentially.
- A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called flowlines.

- Like pseudocode, flowcharts aid in the development and representation of algorithms. Although most programmers prefer pseudocode, flowcharts nicely illustrate how control structures operate.
- The rectangle symbol, also called the action symbol, indicates an action, including a calculation or an input/output operation. Python allows for as many actions as necessary in a sequence structure.
- Perhaps the most important flowchart symbol is the diamond symbol, also called the decision symbol, which indicates a decision is to be performed.
- Python provides three types of selection structures: **`if`**, **`if`**/**`else`** and **`if`**/**`elif`**/**`else`**.
- The **`if`** selection structure either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false.
- The **`if`**/**`else`** selection structure performs an action if a condition is true or performs a different action if the condition is false.
- The **`if`**/**`elif`**/**`else`** selection structure performs one of many different actions, depending on the validity of several conditions.
- The **`if`** selection structure is a single-selection structure—it selects or ignores a single action. The **`if`**/**`else`** selection structure is a double-selection structure—it selects between two different actions. The **`if`**/**`elif`**/**`else`** selection structure is a multiple-selection structure—it selects from many possible actions.
- Python provides two types of repetition structures: **`while`** and **`for`**.
- The words **`if`**, **`elif`**, **`else`**, **`while`** and **`for`** are Python keywords. These keywords are reserved by the language to implement various Python features, such as control structures. Keywords cannot be used as identifiers (e.g., variable names).
- Python has six control structures: sequence, three types of selection and two types of repetition. Each Python program is formed by combining as many control structures of each type as is appropriate for the algorithm the program implements.
- Single-entry/single-exit control structures make it easy to build programs—the control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This is similar to the way a child stacks building blocks; hence, the term control-structure stacking.
- Indentation emphasizes the inherent structure of structured programs and, unlike in most other programming languages, is actually required in Python.
- Nested **`if`**/**`else`** structures test for multiple cases by placing **`if`**/**`else`** selection structures inside other **`if`**/**`else`** selection structures.
- Nested **`if`**/**`else`** structures and the multiple-selection **`if`**/**`elif`**/**`else`** structure are equivalent. The latter form is popular because it avoids deep indentation of the code. Such indentation often leaves little room on a line, forcing lines to be split over multiple lines and decreasing program readability.
- The **`else`** block of the **`if`**/**`elif`**/**`else`** structure is optional. However, most programmers include an **`else`** block at the end of a series of **`elif`** blocks to handle any condition that does not match the conditions specified in the **`elif`** statements. If an **`if`**/**`elif`** statement specifies an **`else`** block, the **`else`** block must be the last block in the statement.
- The **`if`** selection structure can contain several statements in the body of an **`if`** statement, and all these statements must be indented. A set of statements contained within an indented code block is called a suite.
- A fatal logic error causes a program to fail and terminate prematurely. For fatal errors, Python prints an error message called a traceback and exits. A nonfatal logic error allows a program to continue executing, but might produce incorrect results.

- Just as multiple statements can be placed anywhere a single statement can be placed, it is possible to have no statements at all, (i.e., empty statements). The empty statement is represented by placing keyword **`pass`** where a statement normally resides.
- A repetition structure allows the programmer to specify that a program should repeat an action while some condition remains true.
- Counter-controlled repetition uses a variable called a counter to control the number of times a set of statements executes. Counter-controlled repetition often is called definite repetition because the number of repetitions must be known before the loop begins executing.
- A sentinel value (also called a signal value, a dummy value or a flag value) indicates "end of data entry." Sentinel-controlled repetition often is called indefinite repetition because the number of repetitions is not known before the start of the loop.
- In top-down, stepwise refinement, which is essential to the development of well-structured programs, the top is a single statement that conveys the overall function of the program. As such, the top is, in effect, a complete representation of a program. Thus, it is necessary to divide (refine) the top into a series of smaller tasks and list these in the order in which they need to be performed.
- Floating-point numbers contain a decimal point, as in 7.2 or –93.5.
- Dividing two integers results in integer division, in which any fractional part of the calculation is discarded (i.e., truncated).
- To produce a floating-point calculation with integer values, convert one (or both) of the values to a floating-point value with function **`float`**.
- The Python interpreter evaluates expressions in which the data types of the operands are identical. To ensure that the operands are of the same type, the interpreter performs an operation called promotion (also called implicit conversion) on selected operands.
- Python provides several augmented assignment symbols for abbreviating assignment expressions run together.
- Any statement of the form *variable* = *variable operator expression* where operator is a binary operator, such as **`+`**, **`-`**, **`**`**, **`*`**, **`/`**, and **`%`**, can be written in the form *variable operator= expression*.
- Function **`range`** can take one, two or three arguments. If we pass one argument to the function, that argument, called **`end`**, is one greater than the upper bound (highest value) of the sequence.
- If we pass two arguments, the first argument, called **`start`**, is the lower bound—the lowest value in the returned sequence—and the second argument is **`end`**.
- If we pass three arguments, the first two arguments are **`start`** and **`end`**, respectively, and the third argument, called **`increment`**, is the increment value. The sequence produced by a call to **`range`** with an increment value progresses from **`start`** to **`end`** in multiples of the increment value. If **`increment`** is positive, the last value in the sequence is the largest multiple less than **`end`**.
- The increment value of **`range`** also can be negative. In this case, it is a decrement and the sequence produced progresses downwards from **`start`** to **`end`** in multiples of the increment value. The last value in the sequence is the smallest multiple greater than **`end`**.
- The **`break`** statement, when executed in a **`while`** or **`for`** structure, causes immediate exit from that structure. Program execution continues with the first statement after the structure.
- The **`continue`** statement, when executed in a **`while`** or a **`for`** structure, skips the remaining statements in the body of that structure and proceeds with the next iteration of the loop.
- Python provides logical operators that form more complex conditions by combining simple conditions. The logical operators are **`and`** (logical AND), **`or`** (logical OR) and **`not`** (logical NOT, also called logical negation).

- Python evaluates to false or true all expressions that include relational operators and equality operators. A simple condition (e.g., **`age >= 65`**) that is false evaluates to the integer value 0; a simple condition that is true evaluates to the integer value 1. A Python expression that evaluates to the value 0 is false; a Python expression that evaluates to a non-zero integer value is true.
- If a combined condition evaluates to false, the **`and`** operator returns the first value which evaluated to false. Conversely, if the combined condition evaluates to true, the **`and`** operator returns the last value in the condition.
- If a combined condition evaluates to true, the **`or`** operator returns the first value which evaluated to true. Conversely, if the combined condition evaluates to false, the **`or`** operator returns the last value in the condition.
- The **`and`** operator has a higher precedence than the **`or`** operator. Both operators associate from left to right. An expression containing **`and`** or **`or`** operators is evaluated until its truth or falsity is known. This is called short circuit evaluation.
- The **`not`** (logical negation) operator enables a programmer to "reverse" the meaning of a condition. Unlike the **`and`** and **`or`** operators, which combine two conditions (binary operators), the logical negation operator has a single condition as an operand (i.e., **`not`** is a unary operator).

TERMINOLOGY

action/decision model of programming
action symbol
algorithm
`and` (logical AND) operator
augmented addition assignment symbol
augmented assignment statement
augmented assignment symbol
`break` statement
compound statement
connector symbols
`continue` statement
control structure
control-structure nesting
control-structure stacking
counter
counter-controlled repetition
decision symbol
default condition
definite repetition
double-selection structure
diamond symbol
dummy value
empty statement
`end` argument of **`range`** function
exception handling
fatal logic error
first refinement
flag value
`float` function
flowchart
`for` repetition structure
function
`goto` elimination
`goto` statement
`if` selection structure
`if`/**`elif`**/**`else`** selection structure
`if`/**`else`** selection structure
implicit conversion
`increment` argument of **`range`** function
increment value
indefinite repetition
initialization phase
`int` function
keyword
list
logic error
logical negation
logical operator
loop-continuation test
lower bound
multiple-selection structure
nested **`if`**/**`else`** structure
nesting
nesting rule
nonfatal logic error
`not` (logical NOT) operator
off-by-one error
`or` (logical OR) operator
oval symbol
`pass` keyword
procedure
processing phase

program control
promotion
pseudocode
range function
rectangle symbol
repetition structure
second refinement
selection structure
sentinel value
sequence
sequence structure
sequential execution
short-circuit evaluation
signal value
simple condition
single-entry/single-exit control structure
single-selection structure
small circle symbol
stacking rule
start argument of **range** function
structured programming
suite
termination phase
top-down, stepwise refinement
total
traceback
transfer of control
truth table
unary operator
upper bound
while repetition structure

SELF-REVIEW EXERCISES

3.1 Fill in the blanks in each of the following statements:

a) The **if**/**elif**/**else** structure is a __________ structure.
b) The words **if** and **else** are examples of reserved words called Python __________.
c) Sentinel-controlled repetition is called __________ because the number of repetitions is not known before the loop begins executing.
d) The augmented assignment symbol ***=** performs __________.
e) Function __________ creates a sequence of integers.
f) A procedure for solving a problem is called a(n) __________.
g) The keyword __________ represents an empty statement.
h) A set of statements within an indented code block in Python is called a __________.
i) All programs can be written in terms of three control structures, namely, __________, __________ and __________.
j) A __________ is a graphical representation of an algorithm.

3.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Pseudocode is a simple programming language.
b) The **if** selection structure performs an indicated action when the condition is true.
c) The **if**/**else** selection structure is a single-selection structure.
d) A fatal logic error causes a program to execute and produce incorrect results.
e) A repetition structure performs the statements in its body while some condition remains true.
f) Function **float** converts its argument to a floating-point value.
g) The exponentiation operator ****** associates left to right.
h) Function call **range(1, 10)** returns the sequence 1 to 10, inclusive.
i) Sentinel-controlled repetition uses a counter variable to control the number of times a set of instructions executes.
j) The symbol **=** tests for equality.

ANSWERS TO SELF-REVIEW EXERCISES

3.1 a) multiple-selection. b) keywords. c) indefinite repetition. d) multiplication. e) **range**. f) algorithm. g) **pass**. h) suite. i) the sequence structure, the selection structure, the repetition structure. j) flowchart.

3.2 a) False. Pseudocode is an artificial and informal language that helps programmers develop algorithms. b) True. c) False. The **if**/**else** selection structure is a double-selection structure—it selects between two different actions. d) False. A fatal logic error causes a program to terminate. e) True. f) True. g) False. The exponentiation operator associates from right to left. h) False. Function call **range(1, 10)** returns the sequence 1–9, inclusive. i) False. Counter-controlled repetition uses a counter variable to control the number of repetitions; sentinel-control repetition waits for a sentinel value to stop repetition. j) False. The operator **==** tests for equality; the symbol **=** is **for** assignment.

EXERCISES

3.3 Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a Python program that prompts the user to input the miles driven and gallons used for each tankful. The program should calculate and display the miles per gallon obtained for each tankful. After processing all input information, the program should calculate and print the combined miles per gallon obtained for all tankful (= total miles driven divide by total gallons used).

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles / gallon for this tank was 22.421875
Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles / gallon for this tank was 19.417475
Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles / gallon for this tank was 24.000000
Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
```

3.4 A palindrome is a number or a text phrase that reads the same backwards or forwards. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether it is a palindrome. (*Hint*: Use the division and modulus operators to separate the number into its individual digits.)

3.5 Input an integer containing 0s and 1s (i.e., a “binary” integer) and print its decimal equivalent. Appendix C, Number Systems, discusses the binary number system. (*Hint*: Use the modulus and division operators to pick off the “binary” number’s digits one at a time from right to left. Just as in the decimal number system, where the rightmost digit has the positional value 1 and the next digit leftward has the positional value 10, then 100, then 1000, etc., in the binary number system, the rightmost digit has a positional value 1, the next digit leftward has the positional value 2, then 4, then 8, etc. Thus, the decimal number 234 can be interpreted as 2 * 100 + 3 * 10 + 4 * 1. The decimal equivalent of binary 1101 is 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1.)

3.6 The factorial of a nonnegative integer n is written $n!$ (pronounced “n factorial”) and is defined as follows:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ (for values of n greater than or equal to 1)

and

$n! = 1$ (for $n = 0$).

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120. Factorials increase in size very rapidly. What is the largest factorial that your program can calculate before leading to an overflow error?

a) Write a program that reads a nonnegative integer and computes and prints its factorial.

b) Write a program that estimates the value of the mathematical constant *e* by using the formula [*Note*: Your program can stop after summing 10 terms.]

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Write a program that computes the value of e^x by using the formula [*Note*: Your program can stop after summing 10 terms.]

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

3.7 Write a program that prints the following patterns separately, one below the other each pattern separated from the next by one blank line. Use **`for`** loops to generate the patterns. All asterisks (**`*`**) should be printed by a single statement of the form

```
print '*',
```

(which causes the asterisks to print side by side separated by a space). (*Hint*: The last two patterns require that each line begin with an appropriate number of blanks.) Extra credit: Combine your code from the four separate problems into a single program that prints all four patterns side by side by making clever use of nested **`for`** loops. For all parts of this program—minimize the numbers of asterisks and spaces and the number of statements that print these characters.

```
(A)                    (B)                    (C)                    (D)
*                      * * * * * * * * * *    * * * * * * * * * *                      *
* *                    * * * * * * * * *        * * * * * * * * *                    * *
* * *                  * * * * * * * *            * * * * * * * *                  * * *
* * * *                * * * * * * *                * * * * * * *                * * * *
* * * * *              * * * * * *                    * * * * * *              * * * * *
* * * * * *            * * * * *                        * * * * *            * * * * * *
* * * * * * *          * * * *                            * * * *          * * * * * * *
* * * * * * * *        * * *                                * * *        * * * * * * * *
* * * * * * * * *      * *                                    * *      * * * * * * * * *
* * * * * * * * * *    *                                        *    * * * * * * * * * *
```

3.8 *(Pythagorean Triples)* A right triangle can have sides that are all integers. The set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for **`side1`**, **`side2`** and **`hypotenuse`** all no larger than 20. Use a triple-nested **`for`**-loop that tries all possibilities. This is an example of "brute force" computing. You will learn in more advanced computer science courses that there are many interesting problems for which there is no known algorithmic approach other than sheer brute force.

4

Functions

Objectives

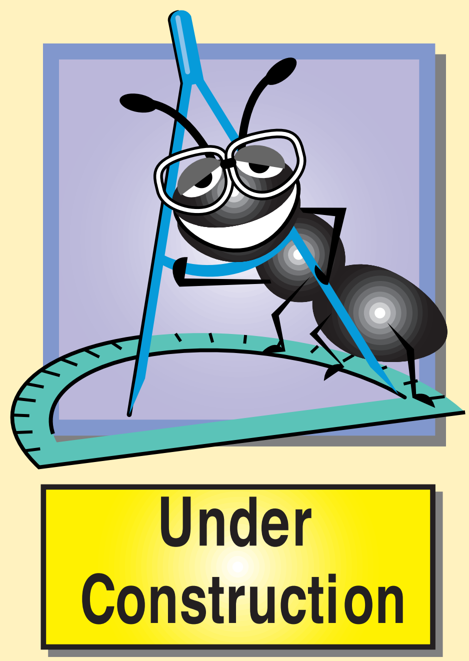
- To understand how to construct programs modularly from small pieces called functions.
- To create new functions.
- To understand the mechanisms of exchanging information between functions.
- To introduce simulation techniques using random number generation.
- To understand how the visibility of identifiers is limited to specific regions of programs.
- To understand how to write and use recursive functions, i.e., functions that call themselves.
- To introduce default and keyword arguments.

Form ever follows function.
Louis Henri Sullivan

E pluribus unum.
(One composed of many.)
Virgil

O! call back yesterday, bid time return.
William Shakespeare
Richard II

When you call me that, smile.
Owen Wister



Outline

4.1 Introduction

Most computer programs that solve real-world problems are larger than the programs presented in the previous chapters. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or *components*, each of which is more manageable than the original program. This technique is called *divide and conquer*. This chapter describes many features of the Python language that facilitate the design, implementation, operation and maintenance of large programs.

4.2 Program Components in Python

Program components in Python are called functions, classes, modules and packages. Typically, Python programs are written by combining *programmer-defined* (programmer-created) functions and classes with functions or classes already available in existing Python modules. A *module* is a file that contains definitions of functions and classes. Many modules can be grouped together into a collection, called a package. In this chapter, we concentrate on functions and we introduce modules and packages; we discuss classes in detail in Chapter 7, Object-Based Programing.

Programmers can define functions to perform specific tasks that execute at various points in a program. These functions are referred to as *programmer-defined* functions. The

actual statements defining the function are written only once, but may be called upon "to do their job" from many points throughout a program. Thus functions are a fundamental unit of *software reuse* in Python because functions allow us to reuse program code.

Python *modules* provide functions that perform such common tasks as mathematical calculations, string manipulations, character manipulations, Web programming, graphics programming and many other operations. These functions simplify the programmer's work, because the programmer does not have to write new functions to perform common tasks. A collection of modules, the *standard library*, is provided as part of the core Python language. These modules are located in the library directory of the Python installation (e.g., **/usr/lib/python2.2** or **/usr/local/lib/python2.2** on Unix/Linux; **\Python\Lib** or **\Python22\Lib** on Windows).

Just as a module groups related definitions, a *package* groups related modules. The package as a whole provides tools to help the programmer accomplish a general task (e.g., graphics or audio programming). Each module in the package defines classes, functions or data that perform specific, related tasks (e.g., creating colors, processing **.wav** files and the like). This text introduces many available Python packages, but creating a robust package is a software engineering exercise beyond the scope of the text.

Good Programming Practice 4.1

Familiarize yourself with the collection of functions and classes in the core Python modules.

Software Engineering Observation 4.1

Avoid "reinventing the wheel". When possible, use standard library module functions instead of writing new functions. This reduces program development time and increases reliability, because you are using well-designed, well-tested code.

Portability Tip 4.1

Using the functions in the core Python modules usually makes programs more portable.

Performance Tip 4.1

Do not try to rewrite existing module functions to make them more efficient. These functions are written to perform well.

A function is *invoked* (i.e., made to perform its designated task) by a *function call*. The function call specifies the function name and provides information (as *arguments*) that the called function needs to perform its job. A common analogy for this is the hierarchical form of management. A boss (the *calling function* or *caller*) requests a worker (the *called function*) to perform a task and *return* (i.e., report back) the results after performing the task. The boss function is unaware of *how* the worker function performs its designated tasks. The worker might call other worker functions, yet the boss is unaware of this decision. We will discuss how "hiding" implementation details promotes good software engineering. Figure 4.1 shows the **boss** function communicating with worker functions **worker1**, **worker2** and **worker3** in a hierarchical manner. Note that **worker1** acts as a boss function to **worker4** and **worker5**. The **boss** function when calling **worker1** need not know about **worker1**'s relationship with **worker4** and **worker5**. Relationships among functions might not always be a hierarchical structure like the one in this figure.

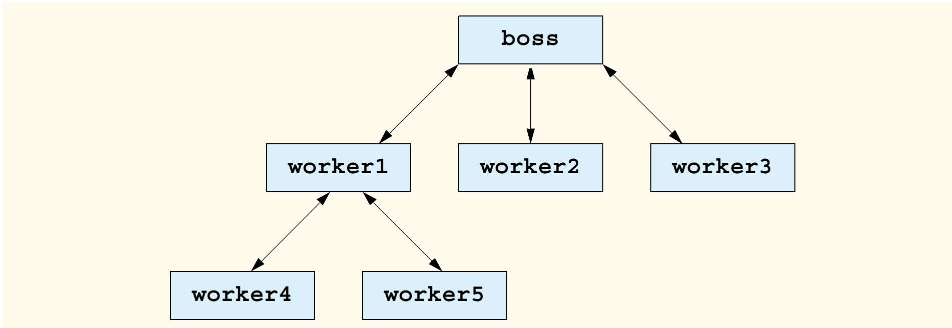


Fig. 4.1 Hierarchical boss-function/worker-function relationship.

4.3 Functions

Functions allow the programmer to modularize a program. All variables created in function definitions are *local variables*—they are known only to the function in which they are declared. Most functions have a list of *parameters* (which are also local variables) that provide the means for communicating information between functions.

There are several motivations for "functionalizing" a program. The divide-and-conquer approach makes program development more manageable. Another motivation is *software reusability*—using existing functions as building blocks for creating new programs. Software reusability is a major benefit of object-oriented programming as we will see in Chapter 7, Object-Based Programming, Chapter 8, Customizing Classes, and Chapter 9, Object-Based Programming: Inheritance. With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than having to write customized code for every task. A third motivation is to avoid repeating code in a program. Packaging code as a function allows the code to be executed in several locations just by calling the function rather than rewriting it in every instance it is used.

Software Engineering Observation 4.2

Each function should be limited to performing a single, well-defined task, and the function name should effectively express that task. This promotes software reusability.

Software Engineering Observation 4.3

If you cannot choose a concise name that expresses a function's task, it is possible that the function is performing too many diverse tasks. Usually, it is best to divide such a function into smaller functions.

4.4 Module `math` Functions

A *module* contains function definitions and other elements (e.g., class definitions) that perform related tasks. The **`math`** module contains functions that allow programmers to perform certain mathematical calculations. We use various **`math`** module functions to introduce the concept of functions and modules. Throughout this text, we discuss many other functions in the core Python modules.

Generally, functions are invoked by writing the name of the function, followed by a left parenthesis, followed by the *argument* (or a comma-separated list of arguments) being

passed to the function, followed by a right parenthesis. To use a function that is defined in a module, a program must *import the module*, using keyword **`import`**. After the module has been imported, the program can invoke functions in that module, using the module's name, a dot (**`.`**) and the function call (i.e., *moduleName*.*functionName*`()`). The interactive session in Fig. 4.2 demonstrates how to print the square root of 900 using the **`math`** module.

When the line

```
print math.sqrt( 900 )
```

executes, the **`math`** module's function **`sqrt`** calculates the square root of the number contained in the parentheses (e.g., 900). The number 900 is the *argument* of the **`math.sqrt`** function. The function *returns* (i.e., gives back as a result) the floating-point value 30.0, which is displayed on the screen.

When the line

```
print math.sqrt( -900 )
```

executes, the function call generates an error, also called an exception, because function **`sqrt`** cannot handle a negative argument. The interpreter displays information about this error to the screen. Exceptions and exception handling are discussed in Chapter 12, Exception Handling.

Common Programming Error 4.1

Failure to import the **`math`** *module when using* **`math`** *module functions is a runtime error. A program must import each module before using its functions and variables.*

Common Programming Error 4.2

When a module is imported via an **`import`** *statement, forgetting to prefix one of its functions with the module name is a runtime error.*

Function arguments can be values, variables or expressions. If **`c1 = 13.0`**, **`d = 3.0`** and **`f = 4.0`**, then the statement

```
print math.sqrt( c1 + d * f )
```

calculates and prints the square root of **`13.0 + 3.0 * 4.0 = 25.0`**, (namely, **`5.0)`**. Some other **`math`** module functions are summarized in Fig. 4.3. (*Note*: Some results are rounded.)

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> import math
>>> print math.sqrt( 900 )
30.0
>>> print math.sqrt( -900 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: math domain error
```

Fig. 4.2 Function **`sqrt`** of module **`math`**.

Method	Description	Example
`acos( x )`	Trigonometric arc cosine of *x* (result in radians)	`acos( 1.0 )` is `0.0`
`asin( x )`	Trigonometric arc sine of *x* (result in radians)	`asin( 0.0 )` is `0.0`
`atan( x )`	Trigonometric arc tangent of *x* (result in radians)	`atan( 0.0 )` is `0.0`
`ceil( x )`	Rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	Trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	Exponential function e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`fabs( x )`	Absolute value of *x*	`fabs( 5.1 )` is `5.1` `fabs( -5.1 )` is `5.1`
`floor( x )`	Rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`fmod( x, y )`	Remainder of *x*/*y* as a floating point number	`fmod( 9.8, 4.0 )` is `1.8`
`hypot( x, y )`	hypotenuse of a triangle with sides of length x and y: sqrt($x^2 + y^2$)	`hypot( 3.0, 4.0 )` is `5.0`
`log( x )`	Natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`log10( x )`	Logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0` `log10( 100.0 )` is `2.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, .5 )` is `3.0`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Fig. 4.3 `math` module functions.

4.5 Function Definitions

Each program we have presented thus far has consisted of a series of statements that sometimes called predefined Python functions to accomplish the program's tasks. We refer to these statements as the *main portion of the program* for the duration of the book, to differentiate it from the part of the program that contains function definitions. We now discuss how programmers write customized functions.

Software Engineering Observation 4.4

In programs containing many functions, the main portion of the program should be implemented as a group of calls to functions that perform the bulk of the program's work.

Consider a program, with a user-defined function **square**, that calculates the squares of the integers from 1 to 10 (Fig. 4.4). Functions must be defined before they are used.

Good Programming Practice 4.2

Place a blank line between function definitions to separate the functions and enhance program readability.

Line 9 of the main program invokes function **square** (defined at lines 5–6) with the statement

```
print square( x ),
```

Function **square** receives a copy of **x** in the parameter **y**.[1] Then **square** calculates **y * y** (line 6). The result is returned to the statement that invoked **square**. The function call (line 9) evaluates to the value returned by the function. This value is displayed by the **print** statement. The value of **x** is not changed by the function call. This process is repeated 10 times using the **for** repetition structure.

The format of a function definition is

def *function-name* **(** *parameter-list* **):**
  statements

where *function-name* is any valid identifier, and *parameter-list* is a comma-separated list of parameter names received by *function-name*. If a function does not receive any values, the parameter list is empty, but the parentheses are still required. The indented statements that follow a **def** statement form the *function body*. The function body is referred to as a *block*.

```
1  # Fig. 4.4: fig04_04.py
2  # Creating and using a programmer-defined function.
3
4  # function definition
5  def square( y ):
6     return y * y
7
8  for x in range( 1, 11 ):
9     print square( x ),
10
11 print
```

```
1 4 9 16 25 36 49 64 81 100
```

Fig. 4.4 Programmer-defined function.

1. Actually, **y** receives a reference to **x**, but **y** behaves as if it were a copy of **x**'s value. This is the concept of pass-by-object-reference, which we introduce in Chapter 5, Lists, Tuples and Dictionaries.

Common Programming Error 4.3

Failure to place a colon (`:`) after a function's parameter list is a syntax error.

Common Programming Error 4.4

*The pair of parentheses `()` in a function call is a Python operator. It causes the function to be called. The function is not invoked if the parentheses are missing from a function call. Normally, control passes through the statement. If a **`print`** statement includes a function call without parentheses, it displays the memory location of the function. If the user intends to assign the result of a function call to a variable, a function call without parentheses binds the function itself to the variable.*

Common Programming Error 4.5

Failure to indent the body of a function is a syntax error.

Good Programming Practice 4.3

It is not advisable to use identical names for the arguments passed to a function and the corresponding parameters in the function definition.

Good Programming Practice 4.4

Choosing meaningful function names and meaningful parameter names ensures program readability and reduces the amount of comments. Writing programs this way creates "self-commenting code."

Software Engineering Observation 4.5

If possible, a function should fit in an editor window. Regardless of the length of a function, it should perform one task well. Small functions promote software reusability.

Testing and Debugging Tip 4.1

Updating a function is easier than updating repeated code throughout a program.

Software Engineering Observation 4.6

Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain and modify.

Software Engineering Observation 4.7

*A function requiring a large number of parameters might be performing too many tasks. Consider dividing the function into smaller functions that perform separate tasks. The function's **`def`** statement should fit on one line, if possible.*

When a function completes its task, the function returns control to the caller. There are three ways to return control to the point from which a function was invoked. If the function does not return a result explicitly, control is returned either when the last indented line is reached or upon execution of the statement

```
return
```

In either case, the function returns **`None`**, a Python value that represents null—indicating that no value has been declared—and evaluates to false in conditional expressions.

If the function does return a result, the statement

return *expression*

returns the value of *expression* to the caller.

Our second example (Fig. 4.5) uses a programmer-defined function, **maximumValue**. This function is independent of the type of its arguments. We use function **maximumValue** to determine and return the largest of three integers, the largest of three floats and the largest of three strings.

Line 15 combines two function calls—**raw_input** and **int**—into one statement. In this case, function **raw_input** reads a value from the user, then the result is passed to function **int** as an argument. The call to function **maximumValue** (line 20) passes the three integers to the programmer-defined function (lines 4–13). The **return** statement in **maximumValue** (line 13) returns the largest integer value to the main program. The **print** statement (line 20) displays the returned value. The same function also returns the maximum float (line 26) and the maximum string (line 32).

```
1  # Fig. 4.5: fig04_05.py
2  # Finding the maximum of three integers.
3
4  def maximumValue( x, y, z ):
5     maximum = x
6
7     if y > maximum:
8        maximum = y
9
10    if z > maximum:
11       maximum = z
12
13    return maximum
14
15 a = int( raw_input( "Enter first integer: " ) )
16 b = int( raw_input( "Enter second integer: " ) )
17 c = int( raw_input( "Enter third integer: " ) )
18
19 # function call
20 print "Maximum integer is:", maximumValue( a, b, c )
21 print   # print new line
22
23 d = float( raw_input( "Enter first float: " ) )
24 e = float( raw_input( "Enter second float: " ) )
25 f = float( raw_input( "Enter third float: " ) )
26 print "Maximum float is: ", maximumValue( d, e, f )
27 print
28
29 g = raw_input( "Enter first string: " )
30 h = raw_input( "Enter second string: " )
31 i = raw_input( "Enter third string: " )
32 print "Maximum string is: ", maximumValue( g, h, i )
```

Fig. 4.5 Programmer-defined **maximum** function. (Part 1 of 2.)

```
Enter first integer: 27
Enter second integer: 12
Enter third integer: 36
Maximum integer is: 36

Enter first float: 12.3
Enter second float: 45.6
Enter third float: 9.03
Maximum float is:  45.6

Enter first string: hello
Enter second string: programming
Enter third string: goodbye
Maximum string is:  programming
```

Fig. 4.5 Programmer-defined **maximum** function. (Part 2 of 2.)

4.6 Random-Number Generation

We now take a brief diversion into a popular programming application—simulation and game playing—to illustrate most of the control structures we have studied. In this and the next section, we develop a game-playing program that incorporates multiple functions.

There is something in the air of a gambling casino that invigorates every type of person from the high-rollers at the plush mahogany-and-felt craps tables to the quarter-poppers at the one-armed bandits. It is the *element of chance*, the possibility that luck will convert a pocketful of money into a mountain of wealth, is what drives scores of people to gambling casinos. The element of chance can be introduced into computer applications through module **random**.

Function ***random.randrange*** generates an integer in the range of its first argument upto, but not including, its second argument. If **randrange** truly produces integers at random, every number in that range has an equal *chance* (or *probability*) of being chosen each time the function is called.

Figure 4.6 displays the results of 20 rolls of a six-sided die to demonstrate module **random**. Function call **random.randrange(1, 7)** produces integers in the range 1–6.

```
# Fig. 4.6: fig04_06.py
# Random integers produced by randrange.

import random

for i in range( 1, 21 ):   # simulates 20 die rolls
   print "%10d" % ( random.randrange( 1, 7 ) ),

   if i % 5 == 0:  # print newline every 5 rolls
      print
```

Fig. 4.6 Random integers produced by **random.randrange(1, 7)**. (Part 1 of 2.)

```
     5          3          3          3          2
     3          2          3          3          4
     2          3          6          5          4
     6          2          4          1          2
```

Fig. 4.6 Random integers produced by **`random.randrange( 1, 7 )`**. (Part 2 of 2.)

To show that these numbers occur with approximately equal likelihood, let us simulate 6000 rolls of a die (Fig. 4.7). Each integer from 1 to 6 should appear approximately 1000 times.

```python
# Fig. 4.7: fig04_07.py
# Roll a six-sided die 6000 times.

import random

frequency1 = 0
frequency2 = 0
frequency3 = 0
frequency4 = 0
frequency5 = 0
frequency6 = 0

for roll in range( 1, 6001 ):       # 6000 die rolls
   face = random.randrange( 1, 7 )

   if face == 1:                    # frequency counted
      frequency1 += 1
   elif face == 2:
      frequency2 += 1
   elif face == 3:
      frequency3 += 1
   elif face == 4:
      frequency4 += 1
   elif face == 5:
      frequency5 += 1
   elif face == 6:
      frequency6 += 1
   else:                             # simple error handling
      print "should never get here!"

print "Face %13s" % "Frequency"
print "   1 %13d" % frequency1
print "   2 %13d" % frequency2
print "   3 %13d" % frequency3
print "   4 %13d" % frequency4
print "   5 %13d" % frequency5
print "   6 %13d" % frequency6
```

Fig. 4.7 Rolling a six-sided die 6000 times. (Part 1 of 2.)

```
Face      Frequency
   1            946
   2           1003
   3           1035
   4           1012
   5            987
   6           1017
```

Fig. 4.7 Rolling a six-sided die 6000 times. (Part 2 of 2.)

As the program output shows, function **random.randrange** simulates the rolling of a six-sided die. Note that program execution should not reach the **else** condition (lines 28–29) provided in the **if**/**elif**/**else** structure, but we provide the condition for good practice.

Testing and Debugging Tip 4.2

Provide a default **else** *case in an* **if/elif/else** *to catch errors even if you absolutely are certain that the program contains no bugs!*

4.7 Example: A Game of Chance

One of the most popular games of chance is a dice game known as "craps," which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> *A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, then that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point." The player loses by rolling a 7 before making the point.*

The program in Fig. 4.8 simulates the game of craps and shows several sample executions.

```
# Fig. 4.8: fig04_08.py
# Craps.

import random

def rollDice():
   die1 = random.randrange( 1, 7 )
   die2 = random.randrange( 1, 7 )
   workSum = die1 + die2
   print "Player rolled %d + %d = %d" % ( die1, die2, workSum )

   return workSum

sum = rollDice()                          # first dice roll

```

Fig. 4.8 Game of craps. (Part 1 of 2.)

```
16  if sum == 7 or sum == 11:                # win on first roll
17     gameStatus = "WON"
18  elif sum == 2 or sum == 3 or sum == 12:  # lose on first roll
19     gameStatus = "LOST"
20  else:                                    # remember point
21     gameStatus = "CONTINUE"
22     myPoint = sum
23     print "Point is", myPoint
24
25  while gameStatus == "CONTINUE":   # keep rolling
26     sum = rollDice()
27
28     if sum == myPoint:             # win by making point
29        gameStatus = "WON"
30     elif sum == 7:                 # lose by rolling 7:
31        gameStatus = "LOST"
32
33  if gameStatus == "WON":
34     print "Player wins"
35  else:
36     print "Player loses"
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 1 + 5 = 6
Point is 6
Player rolled 1 + 6 = 7
Player loses
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 4 = 8
Player rolled 2 + 3 = 5
Player rolled 5 + 4 = 9
Player wins
```

Fig. 4.8 Game of craps. (Part 2 of 2.)

Notice that the player must roll two dice on each roll. Function `rollDice` simulates rolling the dice (lines 6–12). Function `rollDice` is defined once, but it is called from two places in the program (lines 14 and 26). The function takes no arguments, so the parameter list is empty. Function `rollDice` prints and returns the sum of the two dice (lines 10–12).

The game is reasonably involved. The player could win or lose on the first roll or on any subsequent roll. The variable **gameStatus** keeps track of the win/loss status. Variable **gameStatus** is one of the strings **"WON"**, **"LOST"** or **"CONTINUE"**. When the player wins the game, **gameStatus** is set to **"WON"** (lines 17 and 29). When the player loses the game, **gameStatus** is set to **"LOST"** (lines 19 and 31). Otherwise, **gameStatus** is set to **"CONTINUE"**, allowing the dice to be rolled again (line 21).

If the game is won or lost after the first roll, the body of the **while** structure (lines 25–31) is skipped, because **gameStatus** is not equal to **"CONTINUE"** (line 25). Instead, the program proceeds to the **if**/**else** structure (lines 33–36), which prints **"Player wins"** if **gameStatus** equals **"WON"**, but **"Player loses"** if **gameStatus** equals **"LOST"**.

If the game is not won or lost after the first roll, the value of **sum** is assigned to variable **myPoint** (line 22). Execution proceeds with the **while** structure, because **gameStatus** equals **"CONTINUE"**. During each iteration of the **while** loop, **rollDice** is invoked to produce a new **sum** (line 26). If **sum** matches **myPoint**, **gameStatus** is set to **"WON"** (lines 28–29), the **while** test fails (line 25), the **if**/**else** structure prints **"Player wins"** (lines 33–34) and execution terminates. If **sum** is equal to **7**, **gameStatus** is set to **"LOST"** (lines 30–31), the **while** test fails (line 25), the **if**/**else** statement prints **"Player loses"** (lines 35–36) and execution terminates. Otherwise, the **while** loop continues executing.

Note the use of the various program-control mechanisms discussed earlier. The craps program uses one programmer-defined function—**rollDice**—and the **while**, **if**/**else** and **if**/**elif**/**else** structures. The program uses both stacked control structures (the **if**/**elif**/**else** in lines 16–23 and the **while** in lines 25–31) and nested control structures (the **if**/**elif** in lines 28–31 is nested inside the **while** in lines 25–31).

4.8 Scope Rules[2]

Until now, we have not discussed how a Python program stores and retrieves a variable's value. It appears that the value is simply "there" when the program needs it. In fact, Python has strict rules that describe how and when a variable's value can be accessed. These rules are described in terms of *namespaces* and *scopes*. In this section, we discuss how namespaces and scopes affect a program's execution.

We use an example to explain these concepts. Assume that a function contains the following line of code:

```
print x
```

Before a value can be printed to the screen, Python must first find the identifier named **x** and determine the value associated with that identifier. Namespaces store information about an identifier and the value to which it is bound. Python defines three namespaces—local, global and built-in. When a program attempts to access an identifier's value, Python searches the namespaces in a certain order—local, global and built-in namespaces—to see whether and where the identifier exists.

2. Nested scopes are not discussed in this text. Nested scopes are a complex topic and were optional in Python 2.1 but are mandatory in Python 2.2. Information about nested scopes can be found in PEP 227 at **www.python.org/peps/pep-0227.html**.

The first namespace that Python searches is the *local namespace*, which stores bindings created in a block. Function bodies are blocks, so all function parameters and any identifiers the function creates are stored in the function's local namespace. Each function has a unique local namespace—one function cannot access the local namespace of another function. In the example above, Python first searches the function's local namespace for an identifier named **`x`**. If the function's local namespace contains such an identifier, the function prints the value of **`x`** to the screen. If the function's local namespace does not contain an identifier named **`x`** (e.g., the function does not define any parameters or create any identifiers named **`x`**), Python searches the next outer namespace—the *global namespace* (sometimes called the *module namespace*).

The global namespace contains the bindings for all identifiers, function names and class names defined within a module or file. Each module or file's global namespace contains an identifier called **`__name__`** that states the module's name (e.g., **`"math"`** or **`"random"`**). When a Python interpreter session starts or when the Python interpreter begins executing a program stored in a file, the value of **`__name__`** is **`"__main__"`**. In the example above, Python searches for an identifier named **`x`** in the global namespace. If the global namespace contains the identifier (i.e., the identifier was bound to the global namespace before the function was called), Python stops searching for the identifier and the function prints the value of **`x`** to the screen. If the global namespace does not contain an identifier named **`x`**, Python searches the next outer namespace—the *built-in namespace*.

The built-in namespace contains identifiers that correspond to many Python functions and error messages. For example, functions **`raw_input`**, **`int`** and **`range`** belong to the built-in namespace. Python creates the built-in namespace when the interpreter starts, and programs normally do not modify the namespace (e.g., by adding an identifier to the namespace). In the example above, the built-in namespace does not contain an identifier named **`x`**, so Python stops searching and prints an error message stating that the identifier could not be found.

An identifier's *scope* describes the region of a program that can access the identifier's value. If an identifier is defined in the local namespace (e.g., in a function), all statements in the block may access that identifier. Statements that reside outside the block (e.g., in the main portion of a program or in another function) cannot access the identifier. Once the code block terminates (e.g., after a **`return`** statement), all identifiers in that block's local namespace "go out of scope" and are inaccessible.

If an identifier is defined in the global namespace, the identifier has *global scope*. A global identifier is known to all code that executes, from the point at which the identifier is created until the end of the file. Furthermore, if certain criteria are met, functions may access global identifiers. We discuss this issue momentarily. Identifiers contained in built-in namespaces may be accessed by code in programs, modules or functions.

One pitfall that can arise in a program that uses functions is called *shadowing*. When a function creates a local identifier with the same name as an identifier in the module or built-in namespaces, the local identifier shadows the global or built-in identifier. A logic error can occur if the programmer references the local variable when meaning to reference the global or built-in identifier.

Common Programming Error 4.6

Shadowing an identifier in the module or built-in namespace with an identifier in the local namespace may result in a logic error.

Good Programming Practice 4.5

Avoid variable names that shadow names in outer scopes. This can be accomplished by avoiding the use of an identifier with the same name as an identifier in the built-in namespace and by avoiding the use of duplicate identifiers in a program.

Python provides a way for programmers to determine what identifiers are available from the current namespace. Built-in function ***`dir`*** returns a list of these identifiers. Figure 4.9 shows the namespace that Python creates when starting an interactive session. Calling function **`dir`** tells us that the current namespace contains three identifiers: **`__builtins__`**, **`__doc__`** and **`__name__`**. The next command in the session prints the value for identifier **`__name__`**, to demonstrate that this value is **`__main__`** for an interactive session. The subsequent command prints the value for identifier **`__builtins__`**. Notice that we get back a value indicating that this identifier is bound to a module. This indicates that the identifier **`__builtins__`** can be used to refer to the module **`__builtin__`**.We explore this further in Section 4.9. The next command in the interactive session creates a new identifier **`x`** and binds it to the value **`3`**. Calling function **`dir`** again reveals that identifier **`x`** has been added to the session's namespace.

The interactive session in Fig. 4.9 only hints at a Python program's powerful ability to provide information about the identifiers in a program (or interactive session). This is called *introspection*. Python provides many other introspective capabilities, including functions **`globals`** and **`locals`** that return additional information about the global and local namespaces, respectively.

Although functions help make a program easier to debug, scoping issues can introduce subtle errors into a program if the developer is not careful. The program in Fig. 4.10 demonstrates these issues, using global and local variables. Line 4 creates variable **`x`** with the value 1. This variable resides in the global namespace for the program and has global scope. In other words, variable **`x`** can be accessed and changed by any code that appears after line 4. This global variable is shadowed in any function that creates a local variable named **`x`**. In the main program, line 22 prints the value of variable **`x`** (i.e., 1). Lines 24–25 assign the value 7 to variable **`x`** and print its new value.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> print __name__
__main__
>>> print __builtins__
<module '__builtin__' (built-in)>
>>> x = 3    # bind new identifier to global namespace
>>> dir()
['__builtins__', '__doc__', '__name__', 'x']
```

Fig. 4.9 Function **`dir`**.

```
1  # Fig. 4.10: fig04_10.py
2  # Scoping example.
3
4  x = 1 # global variable
5
6  # alters the local variable x, shadows the global variable
7  def a():
8     x = 25
9
10    print "\nlocal x in a is", x, "after entering a"
11    x += 1
12    print "local x in a is", x, "before exiting a"
13
14 # alters the global variable x
15 def b():
16    global x
17
18    print "\nglobal x is", x, "on entering b"
19    x *= 10
20    print "global x is", x, "on exiting b"
21
22 print "global x is", x
23
24 x = 7
25 print "global x is", x
26
27 a()
28 b()
29 a()
30 b()
31
32 print "\nglobal x is", x
```

```
global x is 1
global x is 7

local x in a is 25 after entering a
local x in a is 26 before exiting a

global x is 7 on entering b
global x is 70 on exiting b

local x in a is 25 after entering a
local x in a is 26 before exiting a

global x is 70 on entering b
global x is 700 on exiting b

global x is 700
```

Fig. 4.10 Scopes and keyword **global**.

The program defines two functions that neither receive nor return any arguments. Function **a** (lines 7–12) declares a local variable **x** and initializes it to 25. Then, function **a** prints local variable **x**, increments it and prints it again (lines 10–12). Each time the pro-

gram invokes the function, function **a** recreates local variable **x** and initializes the variable to 25, then increments it to 26.

Function **b** (lines 15–20) does not declare any variables. Instead, line 16 designates **x** as having global scope with keyword ***global***. Therefore, when function **b** refers to variable **x**, Python searches the global namespace for identifier **x**. When the program first invokes function **b** (line 28), the program prints the value of the global variable (7), multiplies the value by 10 and prints the value of the global variable (70) again before exiting the function. The second time the program invokes function **b** (line 30), the global variable contains the modified value, 70. Finally, line 32 prints the global variable **x** in the main program again (700) to show that function **b** has modified the value of this variable.

4.9 Keyword `import` and Namespaces

We have discussed how to import a module and use the functions defined in that module. In this section, we explore how importing a module affects a program's namespace and discuss various ways to import modules into a program.

4.9.1 Importing one or more modules

Consider a program that needs to perform one of the specialized mathematical operations defined in module **math**. The program must first import the module with the line

```
import math
```

The code that imports the module now has a reference to the **math** module in its namespace. After the **import** statement, the program may access any identifiers defined in the **math** module.

The interactive session in Fig. 4.11 demonstrates how an **import** statement affects the session's namespace and how a program can access identifiers defined in a module's namespace. The first line imports the **math** module. The next line then calls function **dir**, to demonstrate that the identifier **math** has been inserted in the session's namespace. As the subsequent **print** statement shows, the identifier is bound to an object that represents the **math** module. If we pass identifier **math** to function **dir**, the function returns a list of all the identifiers in the **math** module's namespace.[3][*Note*: Earlier versions of Python may output different results for **dir()**.]

The next command in the session invokes function **sqrt**. To access an identifier in the **math** module's namespace, we must use the dot (**.**) access operator. The line

```
math.sqrt( 9.0 )
```

first accesses (with the dot access operator) function **sqrt** defined in the **math** module's namespace. The line then invokes (with the parentheses operator) the **sqrt** function, passing an argument of **9.0**.

If a program needs to import several modules, the program can include a separate **import** statement for each module. A program can also import multiple modules in one statement, by separating the module names with commas. Each imported module is added to the program's namespace as demonstrated in the interactive session of Fig. 4.12.

3. Actually, function **dir** returns a list of attributes for the object passed as an argument. In the case of a module, this information amounts to a list of all identifiers (e.g., functions and data) defined in the module.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math']
>>> print math
<module 'math' (built-in)>
>>> dir( math )
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
'cos', 'cosh','e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hy-
pot', 'ldexp', 'log', 'log10','modf', 'pi', 'pow', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
>>> math.sqrt( 9.0 )
3.0
```

Fig. 4.11 Importing a module.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> import math, random
>>> dir()
['__builtins__', '__doc__', '__name__', 'math', 'random']
```

Fig. 4.12 Importing more than one module.

4.9.2 Importing identifiers from a module

In the previous example, we discussed how to access an identifier defined in another module's namespace. To access that identifier, the programmer must use the dot (**.**) access operator. Sometimes, a program uses only one or a few identifiers from a module. In this case, it may be useful to import only those identifiers the program needs. Python provides the ***from/import*** statement to import one or more identifiers from a module directly into the program's namespace.

The interactive session in Fig. 4.13 imports the **sqrt** function directly into the session's namespace. When the interpreter executes the line

```
from math import sqrt
```

the interpreter creates a reference to function **math.sqrt** and places the reference directly into the session's namespace. Now, we can call the function directly without using the dot operator. Just as a program can import multiple modules in one statement, a program can import multiple identifiers from a module in one statement. The line

```
from math import sin, cos, tan
```

imports **math** functions **sin**, **cos** and **tan** directly into the session's namespace. After the **import** statement, a call to function **dir** reveals references to each of these functions.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> from math import sqrt
>>> dir()
['__builtins__', '__doc__', '__name__', 'sqrt']
>>> sqrt( 9.0 )
3.0
>>> from math import sin, cos, tan
>>> dir()
['__builtins__', '__doc__', '__name__', 'cos', 'sin', 'sqrt',
'tan']
```

Fig. 4.13 Importing an identifier from a module.

The interactive session in Fig. 4.14 demonstrates that a program also may import *all* identifiers defined in a module. The statement

```
from math import *
```

imports all identifiers that do not start with an underscore from the **math** module into the interactive session's namespace. Now the programmer can invoke any of the functions from the **math** module, without accessing the function through the dot access operator. However, importing a module's identifiers in this way can lead to serious errors and is considered a dangerous programming practice. Consider a situation in which a program had defined an identifier named **e** and assigned it the string value **"e"**. After executing the preceding **import** statement, identifier **e** is bound to the mathematical floating-point constant *e*, and the previous value for **e** is no longer accessible. In general, a program should never import all identifiers from a module in this way.

Testing and Debugging Tip 4.3

In general, avoid importing all identifiers from a module into the namespace of another module. This method of importing a module should be used only for modules provided by trusted sources, whose documentation explicitly states that such a statement may be used to import the module.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> from math import *
>>> dir()
['__builtins__', '__doc__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'e', 'exp', 'fabs', 'floor',
'fmod', 'frexp', 'hypot', 'ldexp','log', 'log10', 'modf', 'pi',
'pow', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

Fig. 4.14 Importing all identifiers from a module.

4.9.3 Binding names for modules and module identifiers

We have already seen how a program can import a module or specific identifiers from a module. Python's syntax gives the programmer considerable control over how the **import** statement affects a program's namespace. In this section, we discuss this control in more detail and explain further how the programmer can customize the references to imported elements.

The statement

```
import random
```

imports the **random** module and places a reference to the module named **random** in the namespace. In the interactive session in Fig. 4.15, the statement

```
import random as randomModule
```

also imports the **random** module, but the ***as*** clause of the statement allows the programmer to specify the name of the reference to the module. In this case, we create a reference named **randomModule**. Now, if we want to access the **random** module, we use reference **randomModule**.

A program can also use an **import**/**as** statement to specify a name for an identifier that the program imports from a module. The line

```
from math import sqrt as squareRoot
```

imports the **sqrt** function from module **math** and creates a reference to the function named **squareRoot**. The programmer may now invoke the function with this reference.

Typically, module authors use **import**/**as** statements, because the imported element may define names that conflict with identifiers already defined by the author's module. With the **import**/**as** statement, the module author can specify a new name for the imported elements and thereby avoid the naming conflict. Programmers also use the **import**/**as** statement for convenience. A programmer may use the statement to rename a particularly long identifier that the program uses extensively. The programmer specifies a shorter name for the identifier, thus increasing readability and decreasing the amount of typing.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> import random as randomModule
>>> dir()
['__builtins__', '__doc__', '__name__', 'randomModule']
>>> randomModule.randrange( 1, 7 )
1
>>> from math import sqrt as squareRoot
>>> dir()
['__builtins__', '__doc__', '__name__', 'randomModule', 'square-
Root']
>>> squareRoot( 9.0 )
3.0
```

Fig. 4.15 Specifying names for imported elements.

Python's capabilities for importing elements into a program supports component-based programming. The programmer should choose syntax Python appropriate for each situation, keeping in mind that the goal of component-based programming is to create programs that are easier to construct and maintain.

4.10 Recursion

The programs we have discussed thus far generally are structured as functions that call one another in a disciplined, hierarchical manner. For some problems, however, it is useful to have functions call themselves. A *recursive function* is a function that calls itself, either directly or indirectly (through another function). Recursion is an important topic discussed at length in upper-level computer-science courses. In this section and the next, we present simple examples of recursion.

We first consider recursion conceptually and then illustrate several recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or so-called *base case(s)*. If the function is not called with a base case, the function divides the problem into two conceptual pieces—a piece that the function knows how to solve (a base case) and a piece that the function does not know how to solve. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of the original problem. Because this new problem looks like the original problem, the function invokes (calls) a fresh copy of itself to go to work on the smaller problem; this is referred to as a *recursive call* and is also called the *recursion step*. The recursion step normally includes the keyword **`return`**, because this result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the function is still open (i.e., while it has not finished executing). The recursion step can result in many more such recursive calls, as the function divides each new subproblem into two conceptual pieces. For the recursion eventually to terminate, the sequence of smaller and smaller problems must converge on a base case. At that point, the function recognizes the base case and returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original function call eventually returns the final result to the caller. This process sounds exotic when compared with the conventional problem solving techniques we have used to this point. As an example of these concepts at work, let us write a recursive program to perform a popular mathematical calculation.

The factorial of a nonnegative integer n, written $n!$ (and pronounced "n factorial"), is the product

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

with 1! equal to 1, and 0! equal to 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer, **`number`**, greater than or equal to 0 can be calculated iteratively (nonrecursively) using **`for`**, as follows:

```
factorial = 1

for counter in range( 1, number + 1 ):
   factorial *= counter
```

A recursive definition of the factorial function is obtained by observing the following relationship:

$$n! = n \cdot (n - 1)!$$

For example, 5! is clearly equal to 5 * 4!, as is shown by the following equations:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 4.16. Figure 4.16 (a) shows how the succession of recursive calls proceeds until 1! evaluates to 1, which terminates the recursion. Figure 4.16 (b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 4.17 uses recursion to calculate and print the factorials of the integers from 0 to 10. The recursive function **`factorial`** (lines 5–10) first tests to determine whether a terminating condition is true (line 7)—if **`number`** is less than or equal to **`1`** (the base case), **`factorial`** returns 1, no further recursion is necessary and the function terminates. Otherwise, if **`number`** is greater than **`1`**, line 10 expresses the problem as the product of **`number`** and a recursive call to **`factorial`** evaluating the factorial of **`number - 1`**. Note that **`factorial( number - 1 )`** is a simpler version of the original calculation, **`factorial( number )`**.

Common Programming Error 4.7

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

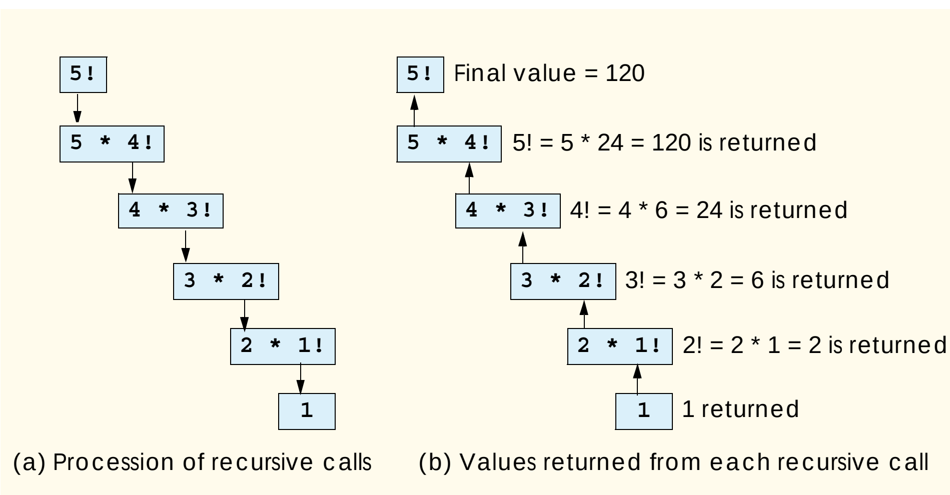


Fig. 4.16 Recursive evaluation of 5!.

```
# Fig. 4.17: fig04_17.py
# Recursive factorial function.

# Recursive definition of function factorial
def factorial( number ):

   if number <= 1:   # base case
      return 1
   else:
      return number * factorial( number - 1 )  # recursive call

for i in range( 11 ):
   print "%2d! = %d" % ( i, factorial( i ) )
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 4.17 Recursive function used to calculate factorials.

4.11 Example Using Recursion: The Fibonacci Series

The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature, in particular, describing a spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…. This number, too, repeatedly occurs in nature and has been called the *golden ratio*, or the *golden mean*. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms, and buildings whose length and width are in the ratio of the golden mean. Postcards often are designed with a golden-mean length/width ratio.

The Fibonacci series can be defined recursively as follows:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci($n - 1$) + fibonacci($n - 2$)

Note that there are two base cases for the Fibonacci calculation—fibonacci(0) is defined to be 0 and fibonacci(1) is defined to be 1. The program of Fig. 4.18 calculates the ith Fibonacci number recursively, using function **`fibonacci`** (lines 4–14). Notice that Fibonacci numbers increase rapidly. Each output box shows a separate execution of the program.

```
# Fig. 4.18: fig04_18.py
# Recursive fibonacci function.

def fibonacci( n ):

   if n < 0:
     print "Cannot find the fibonacci of a negative number."

   if n == 0 or n == 1:   # base case
      return n
   else:

      # two recursive calls
      return fibonacci( n - 1 ) + fibonacci( n - 2 )

number = int( raw_input( "Enter an integer: " ) )
result = fibonacci( number )
print "Fibonacci(%d) = %d" % ( number, result )
```

```
Enter an integer: 0
Fibonacci(0) = 0
```

```
Enter an integer: 1
Fibonacci(1) = 1
```

```
Enter an integer: 2
Fibonacci(2) = 1
```

```
Enter an integer: 3
Fibonacci(3) = 2
```

```
Enter an integer: 4
Fibonacci(4) = 3
```

```
Enter an integer: 6
Fibonacci(6) = 8
```

```
Enter an integer: 10
Fibonacci(10) = 55
```

Fig. 4.18 Recursively generating Fibonacci numbers. (Part 1 of 2.)

```
Enter an integer: 20
Fibonacci(20) = 6765
```

Fig. 4.18 Recursively generating Fibonacci numbers. (Part 2 of 2.)

The initial call to **`fibonacci`** (line 17) is not a recursive call, but all subsequent calls to **`fibonacci`** performed from the body of **`fibonacci`** are recursive. Each time **`fibonacci`** is invoked, it tests for the base case—**`n`** equal to 0 or 1. If this condition is true, **`fibonacci`** returns **`n`** (line 10). Interestingly, if **`n`** is greater than 1, the recursion step generates *two* recursive calls (line 14), each of which is a simpler problem than the original call to **`fibonacci`**. Figure 4.19 illustrates **`fibonacci`** evaluating **`fibonacci( 3 )`**.

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each invocation of the **`fibonacci`** function that does not match one of the base cases (i.e., 0 or 1) results in two more recursive calls to **`fibonacci`**. This set of recursive calls rapidly gets out of hand. Calculating the Fibonacci value of 20 using the program in Fig. 4.18 requires 21,891 calls to the **`fibonacci`** function; calculating the Fibonacci value of 30 requires 2,692,537 calls to the **`fibonacci`** function.

As you try to calculate larger Fibonacci values, you will notice that each consecutive Fibonacci number results in a substantial increase in calculation time and number of calls to the **`fibonacci`** function. For example, the Fibonacci value of 31 requires 4,356,617 calls, and the Fibonacci value of 32 requires 7,049,155 calls. As you can see, the number of calls to fibonacci is increasing quickly—2,692,538 additional calls between Fibonacci values of 31 and 32. This difference in number of calls made between Fibonacci values of 31 and 32 is more than 1.5 times the number of calls for Fibonacci values between 30 and 31. Computer scientists refer to this as *exponential complexity*. Problems of this nature humble even the world's most powerful computers! In the field of complexity theory, computer scientists study how hard algorithms work to complete their tasks. Complexity issues are discussed in detail in the upper-level computer-science course generally called "Algorithms" or "Complexity."

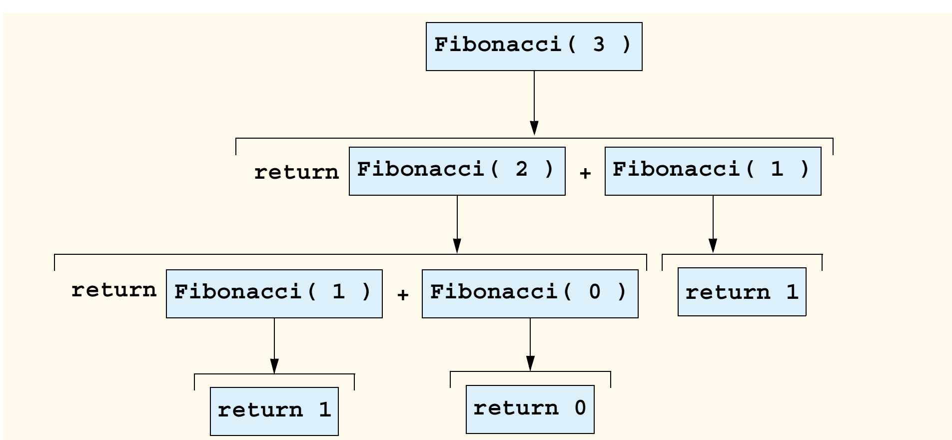


Fig. 4.19 Recursive call to function **`fibonacci`**.

Performance Tip 4.2

Avoid Fibonacci-style recursive programs that result in an exponential "explosion" of calls.

4.12 Recursion vs. Iteration

In the previous sections, we studied two functions that can be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why the programmer might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure (such as **`for`** and **`while`**); recursion uses a selection structure (such as **`if`** and **`if/else`**). Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls. Iteration and recursion both involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion each gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

Recursion has many negatives. It repeatedly invokes the mechanism and, consequently, the overhead of function calls. This repetition can be expensive in both processor time and memory space. Each recursive call causes another copy of the function (actually only the function's variables) to be created; this set of copies can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

Software Engineering Observation 4.8

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach normally is preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Often, a recursive approach can be implemented with few lines of code when a corresponding iterative approach may take large amounts of code. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Performance Tip 4.3

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Common Programming Error 4.8

Accidentally having a function that solves a non-recursive algorithm call itself, either directly or indirectly (through another function), is a logic error.

Let us reconsider some observations that we make repeatedly throughout the book. Good software engineering is important. High performance is important. Unfortunately, these goals are often at odds with one another. Good software engineering is key to making more manageable the task of developing the larger and more complex software sys-

tems. High performance in these systems is key to realizing the systems of the future, which will place ever-greater computing demands on hardware. Where do functions fit in here?

Software Engineering Observation 4.9

Functionalizing programs in a neat, hierarchical manner promotes good software engineering, but it has a price.

Performance Tip 4.4

A heavily functionalized program—as compared with a monolithic (i.e., one-piece) program without functions—makes potentially large numbers of function calls, and these consume execution time and memory space on a computer's processor(s). But monolithic programs are difficult to program, test, debug and maintain.

So functionalize programs judiciously, always keeping in mind the delicate balance between performance and good software engineering.

4.13 Default Arguments

Function calls may commonly pass a particular value of an argument. When defining a function, the programmer can specify an argument as a *default argument*, and the programmer can provide a default value for that argument. Default arguments are a convenience; they allow the programmer to specify fewer arguments when calling a function. When a default argument is omitted in a function call, the interpreter inserts the default value of that argument and passes the argument in the call.

Default arguments must appear to the right of any non-default arguments in a function's parameter list. When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument in the argument list, all arguments to the right of that argument also must be omitted.

Figure 4.20 demonstrates using default arguments in calculating the volume of a box. The function definition for **boxVolume** in line 5 specifies that all three arguments have been given default values of 1. Note that default values should be defined only in the function's **def** statement.

```
1  # Fig. 4.20: fig04_20.py
2  # Using default arguments.
3
4  # function definition with default arguments
5  def boxVolume( length = 1, width = 1, height = 1 ):
6     return length * width * height
7
8  print "The default box volume is:", boxVolume()
9  print "\nThe volume of a box with length 10,"
10 print "width 1 and height 1 is:", boxVolume( 10 )
11 print "\nThe volume of a box with length 10,"
12 print "width 5 and height 1 is:", boxVolume( 10, 5 )
13 print "\nThe volume of a box with length 10,"
14 print "width 5 and height 2 is:", boxVolume( 10, 5, 2 )
```

Fig. 4.20 Default arguments. (Part 1 of 2.)

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 4.20 Default arguments. (Part 2 of 2.)

The first call to **`boxVolume`** (line 8) specifies no arguments and thus uses all three default values. The second call (line 10) passes a **`length`** argument and thus uses default values for the **`width`** and **`height`** arguments. The third call (line 12) passes arguments for **`length`** and **`width`** and thus uses a default value for the **`height`** argument. The last call (line 14) passes arguments for **`length`**, **`width`** and **`height`**, thus using no default values.

Good Programming Practice 4.6

Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments makes programs easier to read.

Common Programming Error 4.9

Default arguments must be the rightmost (trailing) arguments.Omitting an argument other than a rightmost argument is a syntax error.

4.14 Keyword Arguments

The programmer can specify that a function receives one or more *keyword arguments*. The function definition assigns a default value to each keyword. A function may use a default value for a keyword or a function call may assign a new value to the keyword using the format **`keyword = value`**. When using keyword arguments, the position of arguments in the function call is not required to match the position of the corresponding parameters in the function definition. Figure 4.21 demonstrates using keyword arguments in a Python program that displays information about a requested Web site.

```
# Fig. 4.21: fig04_21.py
# Keyword arguments example.

def generateWebsite( name, url = "www.deitel.com",
   Flash = "no", CGI = "yes" ):
   print "Generating site requested by", name, "using url", url

   if Flash == "yes":
      print "Flash is enabled"

```

Fig. 4.21 Keyword parameters. (Part 1 of 2.)

```
11      if CGI == "yes":
12         print "CGI scripts are enabled"
13      print  # prints a new line
14
15   generateWebsite( "Deitel" )
16
17   generateWebsite( "Deitel", Flash = "yes",
18      url = "www.deitel.com/new" )
19
20   generateWebsite( CGI = "no", name = "Prentice Hall" )
```

```
Generating site requested by Deitel using url www.deitel.com
CGI scripts are enabled

Generating site requested by Deitel using url www.deitel.com/new
Flash is enabled
CGI scripts are enabled

Generating site requested by Prentice Hall using url www.deitel.com
```

Fig. 4.21 Keyword parameters. (Part 2 of 2.)

Function **generateWebsite** takes four arguments. The keyword argument names **url**, **Flash** and **CGI** are assigned the default values **"www.deitel.com"**, **"no"** and **"yes"**, respectively (lines 4–5). The function identifies who is requesting the Web site and displays a message if the Web site is Flash- or CGI-enabled (lines 6–13).

The function call in line 15 passes one argument, a value for **name**, to function **generateWebsite**. The function uses the default values given in the definition for the other parameters.

The function call in lines 17–18 passes three arguments to **generateWebsite**. Variable **name** again has the value **"Deitel"**. The call also assigns the value **"yes"** to keyword argument **Flash** and **"www.deitel.com/new"** to keyword argument **url**. This function call illustrates that the order of keyword arguments is more flexible than that of regular arguments in an ordinary function call. The Python interpreter matches the value **"Deitel"** with variable **name** by its position in the function call. The Python interpreter matches the values passed to **url** and **Flash** by their keyword argument names rather than by their positions in the function call. The value of **name** must come first in any call to **generateWebsite** if it is not referenced by specifying a value for **name** in the argument list. Line 20 demonstrates that any function argument can be referenced as a keyword even if it has no default value.

The interactive session of Fig. 4.22 demonstrates common errors when mixing non-keyword and keyword arguments. Function call **test(number1 = "two", "Name")** causes an error, because the non-keyword argument is placed after the keyword argument. Function call **test(number1 = "three")** is incorrect, because function test expects one non-keyword argument.

Common Programming Error 4.10

Misplacing or omitting the value for a non-keyword argument in a function call is an error.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> def test( name, number1 = "one", number2 = "two" ):
...     pass
...
>>> test( number1 = "two", "Name" )
SyntaxError: non-keyword arg after keyword arg
>>> test( number1 = "three" )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: test() takes at least 1 non-keyword argument (0 given)
```

Fig. 4.22 Errors with keyword arguments.

SUMMARY

- Constructing a large program from smaller components, each of which is more manageable than the original program, is a technique called divide and conquer.
- Components in Python are called functions, classes, modules and packages.
- Python programs typically are written by combining new functions and classes the programmer writes with "pre-packaged" functions or classes available in numerous Python modules.
- The programmer can write programmer-defined functions to define specific tasks that could be used at many points in a program.
- A module defines related classes, functions and data. A package groups related modules. The package as a whole provides tools to help the programmer accomplish a general task.
- A function is invoked (i.e., made to perform its designated task) by a function call.
- The function call specifies the function name and provides information (as a comma-separated list of arguments) that the called function needs to do its job.
- All variables created in function definitions are local variables—they are known only in the function in which they are created.
- Most functions have a list of parameters that provide the means for communicating information between functions. A function's parameters are also local variables.
- The divide-and-conquer approach makes program development more manageable.
- Another motivation for using the divide-and-conquer approach is software reusability—using existing functions as building blocks to create new programs.
- A third motivation for using the divide-and-conquer approach is to avoid repeating code in a program. Packaging code as a function allows the code to be executed from several locations in a program simply by calling the function.
- The **`math`** module functions allow the programmer to perform certain common mathematical calculations.
- Functions normally are called by writing the name of the function, followed by a left parenthesis, followed by the argument (or a comma-separated list of arguments) of the function, followed by a right parenthesis.
- To use a function that is defined in a module, a program has to import the module, using keyword **`import`**. After the module has been imported, the program can access a function or a variable in the module, using the module name, a dot (**`.`**) and the function or variable name.

- Functions are defined with keyword **def**.
- The indented statements that follow a **def** statement form the function body. The function body also is referred to as a block.
- There are three ways to return control to the point at which a function was invoked. If the function does not return a result, control is returned simply when the last indented line is reached, or upon executing **return**. If the function does return a result, the statement **return expression** returns the value of **expression** to the caller.
- **None** is a Python value that represents null— indicating that no value has been declared—and that evaluates to false in conditional expressions.
- The element of chance can be introduced into computer applications using module **random**.
- Function **randrange** generates an integer in the range of its first argument to, but not including, its second argument. If **randrange** truly produces integers at random, every number between the first argument and the second argument has an equal chance (or probability) of being chosen each time the function is called.
- Python has strict rules that describe how and when a variable's value can be accessed. These rules are described in terms of namespaces and scopes.
- Namespaces store information about an identifier and the value to which it is bound.
- Python defines three namespaces; when a program attempts to access an identifier's value, Python searches the namespaces in a specific order to see whether and where the identifier exists.
- The local namespace stores bindings created in a block. All function parameters and any identifiers the function creates are stored in the function's local namespace.
- The global (or module) namespace contains the bindings for all identifiers, function names and class names defined in a file or module.
- Each module's global namespace contains an identifier called **__name__** that provides the name for that module. When a Python interpreter session is started or when the Python interpreter is invoked on a program stored in a file, the value of **__name__** is **"__main__"**.
- The built-in namespace contains identifiers that correspond to many Python functions and errors. Python creates the built-in namespace when the interpreter starts, and programs normally do not modify the namespace (e.g., by adding an identifier to the namespace).
- An identifier's scope describes the region of a program that can access the identifier's value.
- If an identifier is defined in the local namespace (e.g., of a function), that identifier has local scope. Once the code block terminates (e.g., when a function returns), all identifiers in that block's local namespace "go out of scope" and no longer can be accessed.
- If an identifier is defined in the global namespace, the identifier has global scope. A global identifier is known to all code that executes within that module, from the point at which the identifier is created until the end of the file.
- When a function creates a local identifier with the same name as an identifier in the module or built-in namespaces, the local identifier is said to shadow the global or built-in identifier. The programmer can introduce a logic error into the program if the programmer refers to the local variable, but intends to refer to the global or built-in identifier.
- A recursive function is a function that calls itself, either directly or indirectly.
- A recursive function actually knows how to solve only the simplest case(s) or so-called base case(s) of a problem.
- If a recursive function is not called with a base case, the function divides the problem into two conceptual pieces: A piece that the function knows how to do (base case), and a piece that the function does not know how to do.

- A recursive function invokes a fresh copy of itself to go to work on a smaller version of the problem; this procedure is referred to as a recursive call and is also called the recursion step.
- Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion also involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.
- Iteration and recursion both involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached.
- Iteration and recursion can both occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.
- Recursion repeatedly invokes the mechanism and, consequently, the overhead of function calls. This can be expensive in both processor time and memory space. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.
- Some function calls commonly pass a particular value of an argument. The programmer can specify that such an argument is a default argument, and the programmer can provide a default value for that argument. When a default argument is omitted in a function call, the interpreter automatically inserts the default value of that argument and passes the argument in the call.
- Default arguments must be the rightmost (trailing) arguments in a function's parameter list. When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument in the argument list, all arguments to the right of that argument also must be omitted.
- The programmer can specify that a function receives one or more keyword arguments. The function definition can assign a value to a keyword argument. Either a function may a default value for a keyword argument or a function call may assign a new value to the keyword argument, using the format **`keyword = value`**.

TERMINOLOGY

`acos` function
`asin` function
`atan` function
base case
built-in namespace
`__builtins__`
calling function
`ceil` function
comma-separated list of arguments
`cos` function
`def` statement
default argument
`dir` function
divide and conquer
dot (**`.`**) operator
`exp` function
expression
`fabs` function
factorial
Fibonacci series
`floor` function
`fmod` function
function
function argument
function body
function call
function definition
function name
function parameter
`global` keyword
global namespace
global variable
`globals` function
`hypot` function

identifier
import keyword
iterative function
keyword argument
local namespace
local variable
locals function
log function
log10 function
"__main__"
main program
math module
module
module namespace
__name__
package
parameter list
probability
random module
randrange function
recursion
recursive function
return keyword
scope
sin function
sqrt function
tan function

SELF-REVIEW EXERCISES

4.1 Fill in the blanks in each of the following statements.

a) Constructing a large program from smaller components is called __________.
b) Components in Python are called __________, __________, __________ and __________.
c) "Pre-packaged" functions or classes are available in Python __________.
d) The __________ module functions allow programmers to perform common mathematical calculations.
e) The indented statements that follow a __________ statement form a function body.
f) The __________ in a function call is the operator that causes the function to be called.
g) The __________ module introduces the element of chance into Python programs.
h) A program can obtain the name of its module through identifier __________.
i) During code execution, three namespaces can be accessed: __________, __________ and __________.
j) A recursive function converges on the __________.

4.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) All variables declared in a function are global to the program containing the function.
b) An **import** statement must be included for every module function used in a program.
c) Function **fmod** returns the floating-point remainder of its two arguments.
d) The keyword **return** displays the result of a function.
e) A function's parameter list is a comma-separated list containing the names of the parameters received by the function when it is called.
f) Function call **random.randrange (1, 7)** produces a random integer in the range 1 to 7, inclusive.
g) An identifier's scope is the portion of the program in which the identifier has meaning.
h) Every call to a recursive function is a recursive call.
i) Omitting the base case in a recursive function can lead to "infinite" recursion.
j) A recursive function may call itself indirectly.

ANSWERS TO SELF-REVIEW EXERCISES

4.1 a) divide and conquer. b) functions, classes, modules, packages. c) modules. d) **math**. e) **def**. f) pair of parentheses. g) **random**. h) **__name__**. i) the local namespace, the global namespace, the built-in namespace. j) base case.

4.2 a) False. All variables declared in a function are local—known only in the function in which they are defined. b) False. Functions included in the `__builtin__` module do not need to be imported. c) True. d) False. Keyword `return` passes control and optionally, the value of an expression, back to the point from which the function was called. e) True. f) False. Function call `random.randrange( 1, 7 )` produces a random integer in the range from 1 to 6, inclusive. g) True. h) False. The initial call to the recursive function is not recursive. i) True. j) True.

EXERCISES

4.3 Implement the following function `fahrenheit` to return the Fahrenheit equivalent of a Celsius temperature.

$$F = \frac{9}{5}C + 32$$

Use this function to write a program that prints a chart showing the Fahrenheit equivalents of all Celsius temperatures 0–100 degrees. Use one position of precision to the right of the decimal point for the results. Print the outputs in a neat tabular format that minimizes the number of lines of output while remaining readable.

4.4 An integer greater than 1 is said to be prime if it is divisible by only 1 and itself. For example, 2, 3, 5 and 7 are prime numbers, but 4, 6, 8 and 9 are not.

a) Write a function that determines whether a number is prime.
b) Use this function in a program that determines and prints all the prime numbers between 2 and 1,000.
c) Initially, you might think that $n/2$ is the upper limit for which you must test to see whether a number is prime, but you need go only as high as the square root of n. Rewrite the program and run it both ways to show that you get the same result.

4.5 An integer number is said to be a perfect number if the sum of its factors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number, because 6 = 1 + 2 + 3. Write a function `perfect` that determines whether parameter `number` is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

4.6 Computers are playing an increasing role in education. The use of computers in education is referred to as *computer-assisted instruction (CAI).* Write a program that will help an elementary school student learn multiplication. Use the random module to produce two positive one-digit integers. The program should then display a question, such as

```
How much is 6 times 7?
```

The student then types the answer. Next, the program checks the student's answer. If it is correct, print the string `"Very good!"` on the screen and ask another multiplication question. If the answer is wrong, display `"No. Please try again."` and let the student try the same question again repeatedly until the student finally gets it right. A separate function should be used to generate each new question. This method should be called once when the program begins execution and each time the user answers the question correctly. (*Hint*: To convert the numbers for the problem into strings for the question, use function `str`. For example, `str( 7 )` returns `"7"`.)

4.7 Write a program that plays the game of "guess the number" as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then displays

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

The player then types a first guess. The program responds with one of the following:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

If the player's guess is incorrect, your program should loop until the player finally gets the number right. Your program should keep telling the player **`Too high`** or **`Too low`** to help the player "zero in" on the correct answer. After a game ends, the program should prompt the user to enter **`"y"`** to play again or **`"n"`** to exit the game.

4.8 *(Towers of Hanoi)* Every budding computer scientist must grapple with certain classic problems. The Towers of Hanoi (see Fig. 4.23) is one of the most famous of these. Legend has it that, in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg, under the constraints that exactly one disk is moved at a time and that at no time may a larger disk be placed above a smaller disk. A third peg is available for holding disks temporarily. Supposedly, the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.

Let us assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving n disks can be viewed in terms of moving only $n - 1$ disks (hence, the recursion), as follows:

a) Move $n - 1$ disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
b) Move the last disk (the largest) from peg 1 to peg 3.
c) Move the $n - 1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving $n = 1$ disk, i.e., the base case. This is accomplished trivially by moving the disk without the need for a temporary holding area.

Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

a) The number of disks to be moved
b) The peg on which these disks are initially threaded
c) The peg to which this stack of disks is to be moved
d) The peg to be used as a temporary holding area

Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

1 → 3 (This means move one disk from peg 1 to peg 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

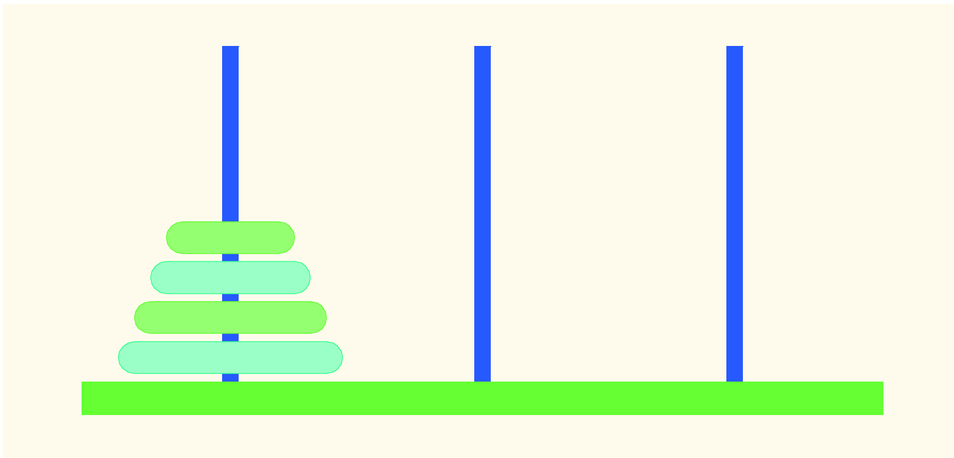


Fig. 4.23 The Towers of Hanoi for the case with 4 disks.

5

Lists, Tuples and Dictionaries

Objectives

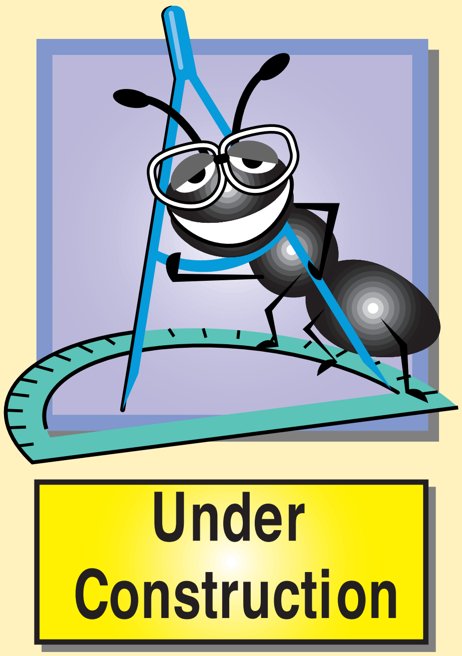
- To understand Python sequences.
- To introduce the list, tuple and dictionary data types.
- To understand how to create, initialize and refer to individual elements of lists, tuples and dictionaries.
- To understand the use of lists to sort and search sequences of values.
- To be able to pass lists to functions.
- To introduce list and dictionary methods.
- To create and manipulate multiple-subscript lists and tuples.

With sobs and tears he sorted out
Those of the largest size …
Lewis Carroll

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.
Robert Herrick

Now go, write it before them in a table,
and note it in a book.
Isaiah 30:8

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
William Shakespeare



Outline

5.1 Introduction

This chapter introduces Python's data-handling capabilities that use *data structures*. Data structures hold and organize information (data). Many types of data structures exist, and each type has features appropriate for certain tasks. *Sequences*, often called *arrays* in other languages, are data structures that store (usually) related data items. Python supports three basic sequence data types: the string, the *list* and the *tuple*. *Mappings*, often called associative arrays or *hashes* in other languages, are data structures that store data in *key-value* pairs. Python supports one mapping data type: the *dictionary*. This chapter discusses Python's sequence and mapping types in the context of several examples. Chapter 22, Data Structures, introduces some high-level data structures (*linked lists*, *queues*, *stacks* and *trees*) that extend Python's basic data types.

5.2 Sequences

A sequence is a series of contiguous values that often are related. We already have encountered sequences in several programs: Python strings are sequences, as is the value returned by function **`range`**—a Python built-in function that returns a list of integers. In this section, we discuss sequences in detail and explain how to refer to a particular *element*, or location, in the sequence.

Figure 5.1 illustrates sequence **`c`**, which contains 12 integer elements. Any element may be referenced by writing the sequence name followed by the element's *position number* in square brackets (**`[]`**). The first element in every sequence is the *zeroth element.* Thus, in sequence **`c`**, the first element is **`c[ 0 ]`**, the second element is **`c[ 1 ]`**, the sixth element of sequence **`c`** is **`c[ 5 ]`**. In general, the *i*th element of sequence **`c`** is **`c[ i - 1 ]`**.

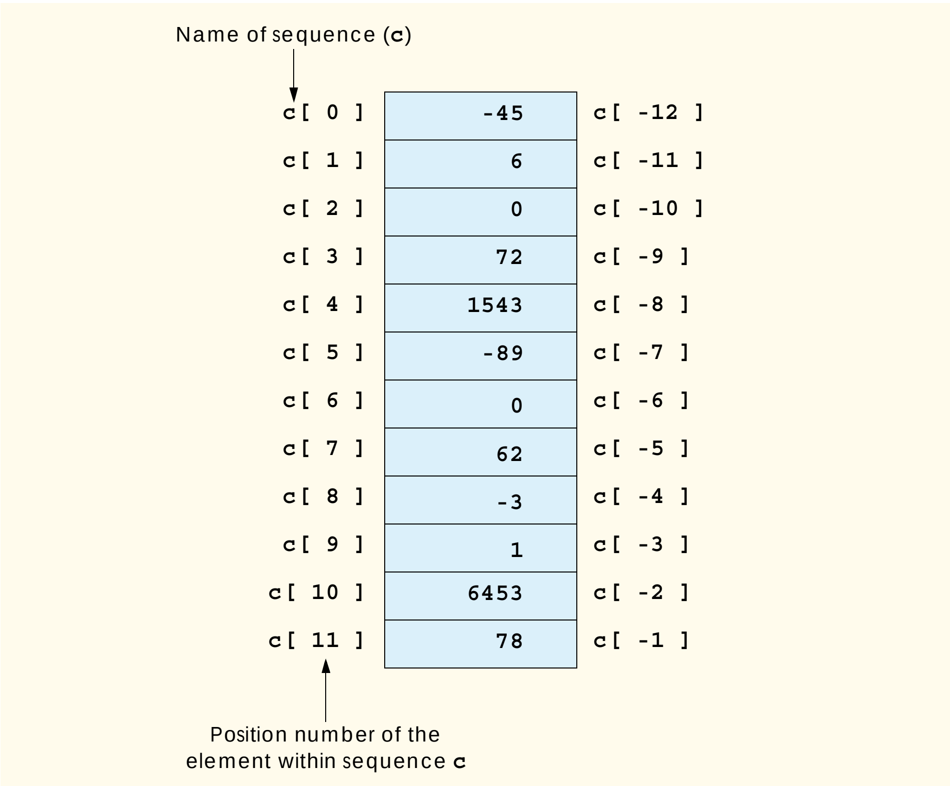


Fig. 5.1 Sequence with elements and indices.

Sequences also can be accessed from the end. The last element is **`c[ -1 ]`**, the second to last element is **`c[ -2 ]`** and the *i*th-from-the-end is **`c[ -i ]`**. Sequences follow the same naming conventions as variables.

The position number more formally is called a *subscript* (or an *index*), which must be an integer or an integer expression. If a program uses an integer expression as a subscript, Python evaluates the expression to determine the index. For example, if variable **`a`** equals **`5`** and variable **`b`** equals **`6`**, then the statement

```
print c[ a + b ]
```

prints the value of **`c[ 11 ]`**. Integer expressions used as subscripts can be useful for iterating over a sequence in a loop.

Python lists and dictionaries are *mutable*—they can be altered. For example, if sequence **`c`** in Fig. 5.1 were mutable, the statement

```
c[ 11 ] = 0
```

modifies the value of element **`11`** by assinging it a new value of **`0`** to replace the original value of **`78`**.

On the other hand, some types of sequences are *immutable*—they cannot be altered (e.g., by changing element values). Python strings and tuples are immutable sequences. For example, if the sequence **c** were immutable, the statement

```
c[ 11 ] = 0
```

would be illegal. Let us examine sequence **c** in detail. The sequence *name* is **c**. The *length* of the sequence is determined by the function call **len(c)**. It is useful to know a sequence's length, because referring to an element outside the sequence results in an "out-of-range" error. Most of the errors discussed in this chapter can be caught as exceptions. [*Note*: We discuss exceptions in Chapter 12, Exception Handling.]

Sequence **c** contains 12 elements, namely **c[0]**, **c[1]**, …, **c[11]**. The range of elements also can be referenced by **c[-12]**, **c[-11]**, ..., **c[-1]**. In this example, **c[0]** contains the *value* **-45**, **c[1]** contains the value **6**, **c[-9]** contains the value **72** and **c[-2]** contains the value **6453**. To calculate the sum of the values contained in the first three elements of sequence **c** and assign the result to variable **sum**, we would write

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ]
```

To divide the value of the seventh element of sequence **c** by 2 and assign the result to the variable **x**, we would write

```
x = c[ 6 ] / 2
```

Common Programming Error 5.1

It is important to note the difference between the "seventh element of the sequence" and "sequence element seven." Sequence subscripts begin at 0, thus the "seventh element of the sequence" has a subscript of 6. On the other hand, "sequence element seven" references subscript 7 (i.e., **c[7]***), which is the eighth element of the sequence. This confusion often leads to "off-by-one" errors.*

Testing and Debugging Tip 5.1

In other programming languages that do not allow negative subscripts, if a negative subscript is accidentally calculated, a run-time error occurs. In Python, such an accidental negative subscript could cause a non-fatal logic error, with the program running to completion and producing invalid results.

The pair of square brackets enclosing the subscript of a sequence is a Python operator. Figure 5.2 shows the precedence and associativity of the operators introduced to this point in the text. They are shown from top to bottom in decreasing order of precedence, with their associativity and types.

5.3 Creating Sequences

Different Python sequences (strings, lists and tuples) require different syntax. We illustrated how Python strings are created by placing the text of the string within quotes. To create an empty string, use a statement like

```
aString = ""
```

Note that we could have used single quotes (**'**) or triple quotes (**"""** or **'''**) to create the string.

Operators	Associativity	Type
`()`	left to right	parentheses
`[]`	left to right	subscript
`.`	left to right	member access
`**`	right to left	exponentiation
`* / // %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== != <>`	left to right	equality

Fig. 5.2 Precedence and associativity of the operators discussed so far.

To create an empty list, use a statement like

```
aList = []
```

To create a list that contains a sequence of values, separate the values by commas inside square brackets (**[]**)

```
aList = [ 1, 2, 3 ]
```

To create an empty tuple, use the statement

```
aTuple = ()
```

To create a tuple that contains a sequence of values, simply separate the values with commas.

```
aTuple = 1, 2, 3
```

Creating a tuple is sometimes referred to as *packing a tuple*. Tuples also can be created by surrounding the comma-separated list of tuple values with optional parentheses. It is the commas that create tuples, not the parentheses.

```
aTuple = ( 1, 2, 3 )
```

When creating a one-element tuple—called a *singleton*—use a statement like

```
aSingleton = 1,
```

Notice that a comma (**,**) follows the value. The comma identifies the variable—**aSingleton**—as a tuple. If the comma were omitted, **aSingleton** would simply contain the integer value 1.

5.4 Using Lists and Tuples

Lists and tuples both contain sequences of values. For example, a list or a tuple may contain the sequence of integers from 1 to 5

```
aList = [ 1, 2, 3, 4, 5 ]
aTuple = ( 1, 2, 3, 4, 5 )
```

In practice, however, Python programmers distinguish between the two data types to represent different kinds of sequences, based on the context of the program. In the next subsections, we discuss the situations for which lists and tuples are best suited.

5.4.1 Using Lists

Although lists are not restricted to homogeneous data types (i.e., values of the same data type), Python programmers typically use lists to store sequences of homogeneous values. For example, either a list may store a sequence of integers that represent test scores or a sequence of strings representing employee names. In general, a program uses a list to store homogeneous values for the purpose of looping over these values and performing the same operation on each value. Usually, the length of the list is not predetermined and may vary over the course of the program. The program in Fig. 5.3 demonstrates how to create, augment and retrieve values from a list.

```
# Fig. 5.3: fig05_03.py
# Creating, accessing and changing a list.

aList = []     # create empty list

# add values to list
for number in range( 1, 11 ):
   aList += [ number ]

print "The value of aList is:", aList

# access list values by iteration
print "\nAccessing values by iteration:"

for item in aList:
   print item,

print

# access list values by index
print "\nAccessing values by index:"
print "Subscript   Value"

for i in range( len( aList ) ):
   print "%9d %7d" % ( i, aList[ i ] )

# modify list
print "\nModifying a list value..."
print "Value of aList before modification:", aList
aList[ 0 ] = -100
aList[ -3 ] = 19
print "Value of aList after modification:", aList
```

Fig. 5.3 List of homogeneous values. (Part 1 of 2.)

```
The value of aList is: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Accessing values by iteration:
1 2 3 4 5 6 7 8 9 10

Accessing values by index:
Subscript   Value
        0       1
        1       2
        2       3
        3       4
        4       5
        5       6
        6       7
        7       8
        8       9
        9      10

Modifying a list value...
Value of aList before modification: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Value of aList after modification: [-100, 2, 3, 4, 5, 6, 7, 19, 9, 10]
```

Fig. 5.3 List of homogeneous values. (Part 2 of 2.)

Line 4 creates empty list, **aList**. Lines 7–8 use a **for** loop to insert the values 1, …, 10 into **aList**, using the **+=** augmented assignment statement. When the value to the left of the **+=** statement is a sequence, the value to the right of the statement also must be a sequence. Thus, line 8 places square brackets around the value to be added to the list. Line 10 prints variable **aList**. Python displays the list as a comma-separated sequence of values inside square brackets. Variable **aList** represents a typical Python list—a sequence containing homogeneous data.

Lines 13–18 demonstrate the most common way of accessing a list's elements. The **for** structure actually iterates over a sequence

```
for item in aList:
```

The **for** structure (lines 15–16) starts with the first element in the sequence, assigns the value of the first element to the control variable (**item**) and executes the body of the **for** loop (i.e., prints the value of the control variable). The loop then proceeds to the next element in the sequence and performs the same operations. Thus, lines 15–16 print each element of **aList**.

List elements also can be accessed through their corresponding indices. Lines 21–25 access each element in **aList** in this manner. The function call in line 24

```
range( len( aList ) )
```

returns a sequence that contains the values **0**, ..., **len(aList) - 1**. This sequence contains all possible element positions for **aList**. The **for** loop iterates through this sequence and, for each element position, prints the position and the value stored at that position.

Lines 30–31 modify some of the list's elements. To modify the value of a particular element, we assign a new value to that element. Line 30 changes the value of the list's first element from 0 to -100; line 31 changes the value of the list's third-from-the-end element from 8 to 19.

If the program attempts to access a nonexistent index (e.g., index 13) in **aList**, the program exits and Python displays an out-of-range error message. The interactive session in Fig. 5.4 demonstrates the results of accessing an out-of-range list element.

Common Programming Error 5.2

Referring to an element outside the sequence is an error.

Testing and Debugging Tip 5.2

When looping through a sequence, the positive sequence subscript should be less than the total number of elements in the sequence (i.e., the subscript should not be larger than the length of the sequence); whereas, the negative sequence subscript should be equal to or greater than the negation of the total number of elements in the sequence. Make sure the loop-terminating condition prevents accessing elements outside this range.

Generally, a program does not concern itself with the length of a list, but simply iterates over the list and performs an operation for each element in the list. Figure 5.5 demonstrates one practical application of using lists in such a manner—creating a *histogram* (a bar graph of frequencies) from a collection of data.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> aList = [ 1 ]
>>> print aList[ 13 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Fig. 5.4 Out-of-range error.

```
# Fig. 5.5: fig05_05.py
# Creating a histogram from a list of values.

values = []   # a list of values

# input 10 values from user
print "Enter 10 integers:"

for i in range( 10 ):
   newValue = int( raw_input( "Enter integer %d: " % ( i + 1 ) ) )
   values += [ newValue ]

```

Fig. 5.5 Histogram created from a list of values. (Part 1 of 2.)

```
13 # create histogram
14 print "\nCreating a histogram from values:"
15 print "%s %10s %10s" % ( "Element", "Value", "Histogram" )
16
17 for i in range( len( values ) ):
18    print "%7d %10d  %s" % ( i, values[ i ], "*" * values[ i ] )
```

```
Enter 10 integers:
Enter integer 1: 19
Enter integer 2: 3
Enter integer 3: 15
Enter integer 4: 7
Enter integer 5: 11
Enter integer 6: 9
Enter integer 7: 13
Enter integer 8: 5
Enter integer 9: 17
Enter integer 10: 1

Creating a histogram from values:
Element      Value Histogram
      0         19  *******************
      1          3  ***
      2         15  ***************
      3          7  *******
      4         11  ***********
      5          9  *********
      6         13  *************
      7          5  *****
      8         17  *****************
      9          1  *
```

Fig. 5.5 Histogram created from a list of values. (Part 2 of 2.)

The program creates an empty list called **values** (line 4). Lines 7–11 input 10 integers from the user and insert those integers into the list. Lines 14–18 create the histogram. For each element in the list, the program prints the element's index and value and a string that contains the same number of asterisks (*****) as the value. The expression

```
"*" * values[ i ]
```

uses the multiplication operator (*****) to create a string with the number of asterisks specified by **values[i]**.

5.4.2 Using Tuples

Whereas lists typically store sequences of homogeneous data, tuples typically store sequences of heterogeneous data—this is a convention, not a rule, that Python programmers follow. Each data item in a tuple provides a part of the total information represented by the tuple. For example, a tuple can represent a student in a class. The tuple could contain the student's name (represented as a string) and age (represented as an integer). Or, a tuple can represent the time of day, using three parts—the hour, minute and second. Although all

these values might be represented as integers, each integer has its own meaning, and the full representation of the time is obtained only by taking all three values together. The length of the tuple (i.e., its number of data items) is predetermined and cannot change during a program's execution.

By convention, each data item in the tuple represents a unique portion of the overall data. Therefore, a program usually does not iterate over a tuple, but accesses the parts of the tuple the program needs to perform its task. Figure 5.6 demonstrates how to create and access a tuple using this idiom.

Lines 5–7 ask the user to enter three integers that represent the hour, minutes and seconds, respectively. Line 9 creates a tuple called **currentTime** to store the user-entered values. Lines 14–16 print the number of seconds that have passed since midnight. We perform a different operation (i.e., multiply each value by a different factor) for each value in the tuple; therefore, the program accesses each value by its index.

As tuples are immutable, Python provides error handling that notifies users when they attempt to modify tuples. For example, if the program attempts to change the first element in **currentTime** to contain the value 0,

```
currentTime[ 0 ] = 0
```

the program exits and Python displays a runtime error

```
Traceback (most recent call last):
  File "fig05_06.py", line 18, in ?
    currentTime[ 0 ] = 0
TypeError: object doesn't support item assignment
```

to indicate that the program illegally attempted to change the value of the immutable tuple.

```
1  # Fig. 5.6: fig05_06.py
2  # Creating and accessing tuples.
3
4  # retrieve hour, minute and second from user
5  hour = int( raw_input( "Enter hour: " ) )
6  minute = int( raw_input( "Enter minute: " ) )
7  second = int( raw_input( "Enter second: " ) )
8
9  currentTime = hour, minute, second   # create tuple
10
11 print "The value of currentTime is:", currentTime
12
13 # access tuple
14 print "The number of seconds since midnight is", \
15    ( currentTime[ 0 ] * 3600 + currentTime[ 1 ] * 60 +
16      currentTime[ 2 ] )
```

```
Enter hour: 9
Enter minute: 16
Enter second: 1
The value of currentTime is: (9, 16, 1)
The number of seconds since midnight is 33361
```

Fig. 5.6 Tuples created and accessed.

Note that the use of lists and tuples introduced in Section 5.4.1 and Section 5.4.2 is not a rule, but rather a convention that Python programmers follow. Python does not limit the data type stored in lists and tuples (i.e., they can contain homogeneous or heterogeneous data). The primary difference between lists and tuples is that lists are mutable whereas tuples are immutable.

5.4.3 Sequence Unpacking

Recall that creating a tuple with

```
aTuple = 1, 2, 3
```

or

```
aTuple = ( 1, 2, 3 )
```

is called packing a tuple, because the values are "packed into" the tuple. Tuples and other sequences also can be *unpacked*—the values stored in the sequence are assigned to various identifiers. Unpacking is a useful programming shortcut for assigning values to multiple variables in a single statement. The program in Fig. 5.7 demonstrates the results of unpacking strings, lists and tuples.

Lines 5–7 create a string, a list and a tuple, each containing three elements. Sequences are unpacked with an assignment statement. The assignment statement in line 11 unpacks the elements in variable **aString** and assigns each element to a variable. The first element is assigned to variable **first**, the second to variable **second** and the third to variable **third**. Line 12 prints the variables to confirm that the string unpacked properly. Lines 14–20 perform similar operations for the elements in variables **aList** and **aTuple**. When unpacking a sequence, the number of variable names to the left of the **=** symbol should equal the number of elements in the sequence to the right of the symbol; otherwise, a runtime error occurs. Notice that when unpacking a sequence, parentheses or brackets are optional to the left of the **=** symbol because there usually are no precedence issues.

```
1  # Fig. 5.7: fig05_07.py
2  # Unpacking sequences.
3
4  # create sequences
5  aString = "abc"
6  aList = [ 1, 2, 3 ]
7  aTuple = "a", "A", 1
8
9  # unpack sequences to variables
10 print "Unpacking string..."
11 first, second, third = aString
12 print "String values:", first, second, third
13
14 print "\nUnpacking list..."
15 first, second, third = aList
16 print "List values:", first, second, third
17
```

Fig. 5.7 Unpacking strings, lists and tuples. (Part 1 of 2.)

```
18 print "\nUnpacking tuple..."
19 first, second, third = aTuple
20 print "Tuple values:", first, second, third
21
22 # swapping two values
23 x = 3
24 y = 4
25
26 print "\nBefore swapping: x = %d, y = %d" % ( x, y )
27 x, y = y, x      # swap variables
28 print "After swapping: x = %d, y = %d" % ( x, y )
```

```
Unpacking string...
String values: a b c

Unpacking list...
List values: 1 2 3

Unpacking tuple...
Tuple values: a A 1

Before swapping: x = 3, y = 4
After swapping: x = 4, y = 3
```

Fig. 5.7 Unpacking strings, lists and tuples. (Part 2 of 2.)

Lines 22–28 demonstrate one benefit of sequence packing and unpacking—swapping the value of two variables. Lines 23–24 create two variables **x** and **y**, with the values 3 and 4, respectively. Line 27

```
x, y = y, x
```

swaps the values assigned to each variable. Python swaps the value by first packing the right-hand side of the statement into a tuple (e.g., **(4, 3)**), then unpacking that tuple to variables **x** and **y**, respectively. Thus, the value assigned to variable **x** is now assigned to variable **y**, and the value assigned to variable **y** is now assigned to variable **x**.

5.4.4 Sequence Slicing

We have discussed how to create sequences and access them through the **[]** operator (to access one element) or a **for** statement (to access all the elements iteratively). Sometimes, a program may need to access a series of sequential values (e.g., the characters of a person's last name in a string that stores the person's full name). For these cases, Python allows programs to *slice* a sequence.

Figure 5.8 demonstrates Python sequence-slicing capabilities. The program creates three sequences—a string, a tuple and a list. The program prompts the user to enter a starting and ending index, creates the specified slice for each sequence and prints the slice to the screen.

```
# Fig. 5.8: fig05_08.py
# Slicing sequences.

```

Fig. 5.8 Sequence slices. (Part 1 of 3.)

```
# create sequences
sliceString = "abcdefghij"
sliceTuple = ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 )
sliceList = [ "I", "II", "III", "IV", "V",
              "VI", "VII", "VIII", "IX", "X" ]

# print strings
print "sliceString: ", sliceString
print "sliceTuple: ", sliceTuple
print "sliceList: ", sliceList
print

# get slices
start = int( raw_input( "Enter start: " ) )
end = int( raw_input( "Enter end: " ) )

# print slices
print "\nsliceString[", start, ":", end, "] = ", \
   sliceString[ start:end ]

print "sliceTuple[", start, ":", end, "] = ", \
   sliceTuple[ start:end ]

print "sliceList[", start, ":", end, "] = ", \
   sliceList[ start:end ]
```

```
sliceString:  abcdefghij
sliceTuple:  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sliceList:  ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII',
'IX', 'X']

Enter start: 3
Enter end: 3

sliceString[ 3 : 3 ] =
sliceTuple[ 3 : 3 ] =  ()
sliceList[ 3 : 3 ] =  []
```

```
sliceString:  abcdefghij
sliceTuple:  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sliceList:  ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII',
'IX', 'X']

Enter start: -4
Enter end: -1

sliceString[ -4 : -1 ] =  ghi
sliceTuple[ -4 : -1 ] =  (7, 8, 9)
sliceList[ -4 : -1 ] =  ['VII', 'VIII', 'IX']
```

Fig. 5.8 Sequence slices. (Part 2 of 3.)

```
sliceString:  abcdefghij
sliceTuple:  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sliceList:  ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII',
'IX', 'X']

Enter start: 0
Enter end: 10

sliceString[ 0 : 10 ] =  abcdefghij
sliceTuple[ 0 : 10 ] =  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sliceList[ 0 : 10 ] =  ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII',
'VIII', 'IX', 'X']
```

Fig. 5.8 Sequence slices. (Part 3 of 3.)

Lines 5–18 create the three sequences and request the user to specify a beginning and ending index for the slice. Lines 21–28 print the specified slice for each sequence. A slice is simply a new sequence, created from an existing sequence. The expression in line 22

```
sliceString[ start:end ]
```

creates (slices) a new sequence from variable **sliceString**. This new sequence contains the values stored at indices **sliceString[start]**, …, **sliceString[end - 1]**. In general, to obtain from *sequence* a slice of the *i*th element through the *j*th element, inclusive, use the expression

sequence`[ `*`i`*`:`*`j`*` + 1 ]`

Figure 5.8 includes three sample outputs from the program. The first sample creates a slice from indices 0 to 10 (e.g., the entire sequence). Recall that the first element in every sequence is the zeroth element. The sequence created from this slice is equivalent to the sequence created with the expression

sequence`[ : ]`

This expression creates a new sequence that is a copy of the original sequence. The above expression is equivalent to the following expressions:

sequence`[ 0 : len( `*sequence*` ) ]`
sequence`[ : len( `*sequence*` ) ]`
sequence`[ 0 : ]`

The syntax for sequence slicing provides a useful shortcut for selecting a portion of an existing sequence. A program can use sequence slicing to create a copy of a list when passing the list to a function. We discuss this issue in Section 5.7 and 5.8.

Note that negative slices cannot access the last element of a list directly (i.e., **sliceString[-4 : -1] = ghi**) because slices apply to points between elements. With negative slices, the last point between elements is the point between elements with indices -2 and -1.

5.5 Dictionaries

In addition to lists and tuples, Python supports another powerful data type, called the *dictionary*. Dictionaries (called *hashes* or *associative arrays* in other languages) are *mapping* constructs consisting of *key-value pairs*. Dictionaries can be thought of as unordered collections of values where each value is referenced through its corresponding key. For example, a dictionary might store phone numbers that can be referenced by a person's name.

The statement

```
emptyDictionary = {}
```

creates an empty dictionary. Notice that curly braces (**{}**) denote dictionaries. To initialize key-value pairs for a dictionary, use the statement

```
dictionary = { 1 : "one", 2 : "two" }
```

Each key-value pair is of the form

key **:** *value*

A comma separates each key-value pair. Dictionary keys must be immutable values, such as strings, numbers or tuples. Dictionary values can be of any Python data type.

Common Programming Error 5.3

Using a list or a dictionary for a dictionary key is an syntax error.

Figure 5.9 demonstrates how to create, initialize, access and manipulate simple dictionaries. Lines 5–6 create and print an empty dictionary. Line 9 creates a dictionary **grades** and initializes the dictionary to contain four key-value pairs. The keys are strings that contain student names, and the integer values represent the students' grades. Line 10 prints the value assigned to variable **grades**. Observe that the application displays **grades** in a different order than the declaration; this is because a dictionary is an unordered collection of key-value pairs. Also, notice in the output that the dictionary keys appear in single quotes, because Python displays strings in single quotes.

```
1  # Fig. 5.09: fig05_09.py
2  # Creating, accessing and modifying a dictionary.
3
4  # create and print an empty dictionary
5  emptyDictionary = {}
6  print "The value of emptyDictionary is:", emptyDictionary
7
8  # create and print a dictionary with initial values
9  grades = { "John": 87, "Steve": 76, "Laura": 92, "Edwin": 89 }
10 print "\nAll grades:", grades
11
12 # access and modify an existing dictionary
13 print "\nSteve's current grade:", grades[ "Steve" ]
14 grades[ "Steve" ] = 90
15 print "Steve's new grade:", grades[ "Steve" ]
```

Fig. 5.9 Dictionaries created, accessed and modified. (Part 1 of 2.)

```
16
17  # add to an existing dictionary
18  grades[ "Michael" ] = 93
19  print "\nDictionary grades after modification:"
20  print grades
21
22  # delete entry from dictionary
23  del grades[ "John" ]
24  print "\nDictionary grades after deletion:"
25  print grades
```

```
The value of emptyDictionary is: {}

All grades: {'Edwin': 89, 'John': 87, 'Steve': 76, 'Laura': 92}

Steve's current grade: 76
Steve's new grade: 90

Dictionary grades after modification:
{'Edwin': 89, 'Michael': 93, 'John': 87, 'Steve': 90, 'Laura': 92}

Dictionary grades after deletion:
{'Edwin': 89, 'Michael': 93, 'Steve': 90, 'Laura': 92}
```

Fig. 5.9 Dictionaries created, accessed and modified. (Part 2 of 2.)

Line 13 accesses a particular dictionary value, using the **[]** operator. Dictionary values are accessed with the expression

dictionaryName **[** *key* **]**

In line 13, the *dictionaryName* is **grades** and the *key* is the string **"Steve"**. This expression evaluates to the value stored in the dictionary at key **"Steve"**, namely, **76**. Line 14 assigns a new value, **90**, to the key **"Steve"**. Dictionary values are modified using syntax similar to that of modifying lists. Line 15 prints the result of changing the dictionary value.

Line 18 inserts a new key-value pair into the dictionary. Although this statement resembles the syntax for modifying an existing dictionary value, it inserts a new key-value pair because **Michael** is a new key. The statement

dictionaryName **[** *key* **] =** *value*

modifies the *value* associated with *key*, if the dictionary already contains that key. Otherwise, the statement inserts the key-value pair into the dictionary.

Software Engineering Observation 5.1

When adding a key-value pair to a dictionary, mis-typing the key could be a source of inadvertent errors.

Lines 19–20 print the results of adding a new key-value pair to the dictionary. The order in which the key-value pairs are printed is entirely arbitrary (remember that a dictionary is an unordered collection of key-value pairs).

The expression *dictionaryName* `[` *key* `]` can lead to subtle programming errors. If this expression appears on the left-hand side of an assignment statement and the dictionary does not contain the key, the assignment statement inserts the key-value pair into the dictionary. However, if the expression appears to the right of an assignment statement (or any statement that simply attempts to access the value stored at the specified key), then the statement causes the program to exit and to display an error message, because the program is trying to access a nonexistent key.

Common Programming Error 5.4

Attempting to access a nonexistent dictionary key is a "key error", a runtime error.

Line 23 deletes an entry from the dictionary. The statement

`del` *dictionaryName* `[` *key* `]`

removes the specified key and its value from the dictionary. If the specified key does not exist in the dictionary, then the above statement causes the program to exit and to display an error message. Again, this is because the program is accessing a nonexistent key. This runtime error can be caught through exception handling, which we discuss in Chapter 12.

Dictionaries are powerful data types that help programmers accomplish sophisticated tasks. Many Python modules provide data types similar to dictionaries that facilitate access and manipulation of more complex data. In the next section, we explore the dictionary's capabilities further.

5.6 List and Dictionary Methods

We have seen how sequences and dictionaries enable programmers to accomplish high-level data manipulation, such as storing and retrieving data. We now introduce a new programming concept, the *method*, to extend data-manipulation capabilities.

As discussed in Chapter 2, Introduction to Python Programming, all Python data types contain at least three properties: a value, a type and a location. Some Python data types (e.g., strings, lists and dictionaries) also contain methods. A method is a function that performs the behaviors (tasks) of an object. In this section, we discuss list and dictionary methods; we discuss string methods in Chapter 13, Strings Manipulation and Regular Expressions.

List methods implement several behaviors, such as appending a value to the end of a list or determining the index of a particular element in the list. The program of Fig. 5.10 appends items to the end of a list, using a list method. The program asks the user to enter the names of Shakespearean plays and appends the names to a list. Line 4 creates an empty list, ***playList***, to store the names of the plays entered by the user. The **`for`** structure (lines 8–10) uses list method ***append*** to append items to the end of variable **`playList`**. Method **`append`** takes as an argument the new element to insert at the end of the list. To invoke the list method, specify the name of the list, followed by the dot (**`.`**) access operator, followed by the method call (i.e., method name and necessary arguments). Lines 14–15 define another **`for`** loop that prints the names of the user-entered Shakespearean plays. Notice that line 15 uses the **`-`** formatting character to left align the names.

Figure 5.10 demonstrates how a data type's methods provide a way for programmers to create applications that perform useful data-manipulation tasks. Figure 5.11 uses another list method to perform a more typical data-manipulation task—counting the number of times a

```
# Fig. 5.10: fig05_10.py
# Appending items to a list.

playList = []      # list of favorite plays

print "Enter your 5 favorite Shakespearean plays.\n"

for i in range( 5 ):
   playName = raw_input( "Play %d: " % ( i + 1 ) )
   playList.append( playName )

print "\nSubscript     Value"

for i in range( len( playList ) ):
   print "%9d     %-25s" % ( i + 1, playList[ i ] )
```

```
Enter your 5 favorite Shakespearean plays.

Play 1: Richard III
Play 2: Henry V
Play 3: Twelfth Night
Play 4: Hamlet
Play 5: King Lear

Subscript     Value
        1     Richard III
        2     Henry V
        3     Twelfth Night
        4     Hamlet
        5     King Lear
```

Fig. 5.10 Appending items to a list.

particular value occurs in a list. Lines 4–7 create a list (**responses**) that contains several values between 1–10. Lines 11–12 contain a **for** loop that calls list method ***count*** to return the amount of times an element appears in a list. Method **count** takes as an argument a value of any data type. If the list contains no elements with the specified value, method **count** returns 0. Lines 11–12 print the frequency of each value in the list.

```
1  # Fig. 5.11: fig05_11.py
2  # Student poll program.
3
4  responses = [ 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
5                1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
6                6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
7                5, 6, 7, 5, 6, 4, 8, 6, 8, 10 ]
8
9  print "Rating     Frequency"
10
11 for i in range( 1, 11 ):
12    print "%6d %13d" % ( i, responses.count( i ) )
```

Fig. 5.11 List method **count**. (Part 1 of 2.)

```
Rating     Frequency
     1             2
     2             2
     3             2
     4             2
     5             5
     6            11
     7             5
     8             7
     9             1
    10             3
```

Fig. 5.11 List method **count**. (Part 2 of 2.)

Lists provide several other useful methods. Figure 5.12 summarizes these methods. Throughout the text, we create programs that invoke list methods to accomplish tasks.

Method	Purpose
append(*item* **)**	Inserts *item* at the end of the list.
count(*element* **)**	Returns the number of occurrences of *element* in the list.
extend(*newList* **)**	Inserts the elements of *newList* at the end of the list.
index(*element* **)**	Returns the index of the first occurrence of *element* in the list. If element is not in the list, a **ValueError** exception occurs. [*Note*: We discuss exceptions in Chapter 12, Exception Handling.]
insert(*index, item* **)**	Inserts *item* at position *index*.
pop(*[index]* **)**	Parameter index is optional. If this method is called without arguments, it removes and returns the last element in the list. If parameter *index* is specified, this method removes and returns the element at position *index*.
remove(*element* **)**	Removes the first occurrence of *element* from the list. If *element* is not in the list, a **ValueError** exception occurs.
reverse()	Reverses the contents of the list in place (rather than creating a reversed copy).
sort(*[compare-function]* **)**	Sorts the content of the list in place. The optional parameter *compare-function* is a function that specifies the compare criteria. The *compare-function* takes any two elements of the list (x and y) and returns -1 if x should appear before y, 0 if the orders of x and y do not matter and 1 if x should appear after y. [*Note*: We discuss sorting in Section 5.9.]

Fig. 5.12 List methods.

The dictionary data type also provides many methods that enable the programmer to manipulate the stored data. Figure 5.13 demonstrates three dictionary methods. Lines 4–7 create the dictionary **monthsDictionary** that represents the months of the year. Line 10 uses dictionary method ***items*** to print the dictionary's key-value pairs to the screen. The method returns a list of tuples, where each tuple contains a key-value pair.

```
1  # Fig. 5.13: fig05_13.py
2  # Dictionary methods.
3
4  monthsDictionary = { 1 : "January", 2 : "February", 3 : "March",
5                      4 : "April", 5 : "May", 6 : "June", 7 : "July",
6                      8 : "August", 9 : "September", 10 : "October",
7                      11 : "November", 12 : "December" }
8
9  print "The dictionary items are:"
10 print monthsDictionary.items()
11
12 print "\nThe dictionary keys are:"
13 print monthsDictionary.keys()
14
15 print "\nThe dictionary values are:"
16 print monthsDictionary.values()
17
18 print "\nUsing a for loop to get dictionary items:"
19
20 for key in monthsDictionary.keys():
21    print "monthsDictionary[", key, "] =", monthsDictionary[ key ]
```

```
The dictionary items are:
[(1, 'January'), (2, 'February'), (3, 'March'), (4, 'April'), (5,
'May'), (6, 'June'), (7, 'July'), (8, 'August'), (9, 'September'), (10,
'October'), (11, 'November'), (12, 'December')]

The dictionary keys are:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

The dictionary values are:
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'Au-
gust', 'September', 'October', 'November', 'December']

Using a for loop to get dictionary items:
monthsDictionary[ 1 ] = January
monthsDictionary[ 2 ] = February
monthsDictionary[ 3 ] = March
monthsDictionary[ 4 ] = April
monthsDictionary[ 5 ] = May
monthsDictionary[ 6 ] = June
monthsDictionary[ 7 ] = July
monthsDictionary[ 8 ] = August
monthsDictionary[ 9 ] = September
monthsDictionary[ 10 ] = October
monthsDictionary[ 11 ] = November
monthsDictionary[ 12 ] = December
```

Fig. 5.13 Dictionary methods **items**, **keys** and **values**.

Dictionary method ***keys*** (line 13) returns an unordered list of the dictionary's keys. Similarly, dictionary method ***values*** (line 16) returns an unordered list of the dictionary's values. Lines 20–21 demonstrate a common use of dictionary method **keys**. The **for** loop iterates over the dictionary keys. Each key is assigned to control variable **key**. Line 21 prints both the key and the value associated with that key. Figure 5.14 summarizes the dictionary methods.

Method	Description
clear()	Deletes all items from the dictionary.
copy()	Creates and returns a shallow copy of the dictionary (the elements in the new dictionary are references to the elements in the original dictionary).
get(*key [, returnValue]* **)**	Returns the value associated with *key*. If *key* is not in the dictionary and if *returnValue* is specified, returns the specified value. If *returnValue* is not specified, returns **None**.
has_key(*key* **)**	Returns **1** if *key* is in the dictionary; returns **0** if *key* is not in the dictionary.
items()	Returns a list of tuples that are key-value pairs.
keys()	Returns a list of keys in the dictionary.
popitem()	Removes and returns an arbitrary key-value pair as a tuple of two elements. If dictionary is empty, a **KeyError** exception occurs. [*Note*: We discuss exceptions in Chapter 12, Exception Handling.] This method is useful for accessing an element (i.e., print the key-value pair) before removing it from the dictionary.
setdefault(*key* [, *dummyValue*] **)**	Behaves similarly to method **get**. If key is not in the dictionary and *dummyValue* is specified, inserts the key and the specified value into dictionary. If *dummyValue* is not specified, value is **None**.
update(*newDictionary* **)**	Adds all key-value pairs from *newDictionary* to the current dictionary and overrides the values for keys that already exist.
values()	Returns a list of values in the dictionary.
iterkeys()	Returns an iterator of dictionary keys. [*Note*: We discuss iterators in Appendix O, Additional Python 2.2 Features.]
iteritems()	Returns an iterator of key-value pairs. [*Note*: We discuss iterators in Appendix O, Additional Python 2.2 Features.]
itervalues()	Returns an iterator of dictionary values. [*Note*: We discuss iterators in Appendix O, Additional Python 2.2 Features.]

Fig. 5.14 Dictionary methods.

Dictionary method **copy** returns a new dictionary that is a *shallow* copy of the original dictionary. In a shallow copy, the elements in the new dictionary are references to the elements in the original dictionary.

The interactive session in Fig. 5.15 demonstrates the difference between shallow and *deep* copies. We first create **dictionary**, which contains one value—a list of numbers. We then invoke dictionary method **copy** to create a shallow copy of **dictionary**, and we assign the copy to variable **shallowCopy**. The values stored for key **"listKey"** in both dictionaries reference the same object. To underscore this fact, we insert the value 4 at the end of the list stored in **dictionary**. We then print the value of variables **dictionary** and **shallowCopy**. Notice that the list has been changed in both copies of the dictionary. This is a consequence of doing a shallow copy, which does not create a fully independent copy of the original dictionary.

Sometimes, a shallow copy is sufficient for a program, especially if the dictionaries contain no references to other Python objects (i.e., they contain only literal numeric values or immutable values). However, sometimes it is necessary to create a copy—called a *deep copy*—that is independent of the original dictionary. To create a deep copy, Python provides module ***copy***. The remainder of the interactive session in Fig. 5.15 creates a deep copy of variable **dictionary**. We first import function ***deepcopy*** from module **copy**. We then call **deepcopy** and pass **dictionary** as an argument. The function call returns a deep copy of **dictionary**, and we assign the copy to variable **deepCopy**. The value associated with **deepCopy["listKey"]** is now independent of the value associated with that key in variables **dictionary** and **shallowCopy**. To demonstrate this fact, we append a new value to **dictionary**'s list and print the values for **dictionary**, **shallowCopy** and **deepCopy**.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> dictionary = { "listKey" : [ 1, 2, 3 ] }
>>> shallowCopy = dictionary.copy()        # make a shallow copy
>>> dictionary[ "listKey" ].append( 4 )
>>> print dictionary
{'listKey': [1, 2, 3, 4]}
>>> print shallowCopy
{'listKey': [1, 2, 3, 4]}

>>> from copy import deepcopy
>>> deepCopy = deepcopy( dictionary )      # make a deep copy
>>> dictionary[ "listKey" ].append( 5 )
>>> print dictionary
{'listKey': [1, 2, 3, 4, 5]}
>>> print shallowCopy
{'listKey': [1, 2, 3, 4, 5]}
>>> print deepCopy
{'listKey': [1, 2, 3, 4]}
```

Fig. 5.15 Difference between a shallow copy and a deep copy.

Shallow and deep copies reflect how Python handles references (i.e., names of objects). The programmer should exercise caution when dealing with references to objects like lists and dictionaries, because changing an object affects the value of all the names that refer to that object. In the next two sections, we discuss how passing a reference to a function affects an object's value.

Software Engineering Observation 5.2

`deepCopyList = originalList[:]` does a deep copy which means that the `deepCopyList` is a deep copy of the `originalList`.

5.7 References and Reference Parameters

To perform tasks, functions require certain input values, which the main program or functions have (or know). The main program (e.g., a program that simulates a calculator) may ask users for input, and those input values are sent, in turn, to functions (e.g., **`add`**, **`subtract`**). The values, or arguments, have to be passed to the functions through a certain protocol. In many programming languages, the two ways to pass arguments to functions are *pass-by-value* and *pass-by-reference*. When an argument is passed by value, a copy of the argument's value is made and passed to the called function.

Testing and Debugging Tip 5.3

With pass-by-value, changes to the called function's copy do not affect the original variable's value in the calling code. This prevents accidental side effects that can hinder the development of correct and reliable software systems.

With pass-by-reference, the caller allows the called function to access the caller's data directly and to modify that data. Pass-by-reference can improve performance by eliminating the overhead of copying large amounts of data. However, pass-by-reference can weaken security, because the called function can access the caller's data.

Unlike many other languages, Python does not allow programmers to choose between pass-by-value and pass-by-reference when passing arguments. Python arguments are always *passed by object reference*—the function receives references to the values passed as arguments. In practice, pass-by-object-reference can be thought of as a combination of pass-by-value and pass-by-reference. If a function receives a reference to a mutable object (e.g., a dictionary or a list), the function can modify the original value of the object. It is as if the object had been passed by reference. If a function receives a reference to an immutable object (e.g., a number, a string or a tuple, whose elements are immutable values), the function cannot modify the original object directly. It is as if the object had been passed by value.

As always, it is important for the programmer to be aware of when an object may be modified by the function to which it is passed. Remembering the preceding rules and understanding how Python treats references to objects is essential to creating large and sophisticated Python systems.

5.8 Passing Lists to Functions

In this section, we discuss references further by examining what happens when a program passes a list to a function. The results we discover hold true for other mutable Python objects, such as dictionaries. To pass a list argument to a function, specify the name of the list without square brackets. For example, if list **`hourlyTemperatures`** has been created as

```
hourlyTemperatures = [ 39, 43, 45 ]
```

the function call

```
modifyList( hourlyTemperatures )
```

passes list **hourlyTemperatures** to function **modifyList**.

Although entire lists can be changed by a function, individual list elements that are numeric or immutable sequence data types cannot be changed. To pass a list element to a function, use the subscripted name of the list element as an argument in the function call.

The program of Fig. 5.16 demonstrates the difference between passing an entire list and passing a list element. Line 12 creates variable **aList**. The **for** loop at lines 17–18 prints the items of the list. Line 20 invokes function **modifyList** and passes the function variable **aList**. Function **modifyList** (lines 4–7) multiplies each element by 2. To illustrate that **aList**'s elements are modified, the **for** loop at lines 24–25 displays the list elements again. As the output shows, the elements of **aList** were modified by **modifyList**.

```
1  # Fig. 5.16: fig05_16.py
2  # Passing lists and individual list elements to functions.
3
4  def modifyList( aList ):
5
6     for i in range( len( aList ) ):
7        aList[ i ] *= 2
8
9  def modifyElement( element ):
10    element *= 2
11
12 aList = [ 1, 2, 3, 4, 5 ]
13
14 print "Effects of passing entire list:"
15 print "The values of the original list are:"
16
17 for item in aList:
18    print item,
19
20 modifyList( aList )
21
22 print "\n\nThe values of the modified list are:"
23
24 for item in aList:
25    print item,
26
27 print "\n\nEffects of passing list element:"
28 print "aList[ 3 ] before modifyElement:", aList[ 3 ]
29 modifyElement( aList[ 3 ] )
30 print "aList[ 3 ] after modifyElement:", aList[ 3 ]
31
32 print "\nEffects of passing slices of list:"
33 print "aList[ 2:4 ] before modifyList:", aList[ 2:4 ]
34 modifyList( aList[ 2:4 ] )
35 print "aList[ 2:4 ] after modifyList:", aList[ 2:4 ]
```

Fig. 5.16 Passing lists and individual list elements to methods. (Part 1 of 2.)

```
Effects of passing entire list:
The values of the original list are:
1 2 3 4 5

The values of the modified list are:
2 4 6 8 10

Effects of passing list element:
aList[ 3 ] before modifyElement: 8
aList[ 3 ] after modifyElement: 8

Effects of passing slices of list:
aList[ 2:4 ] before modifyList: [6, 8]
aList[ 2:4 ] after modifyList: [6, 8]
```

Fig. 5.16 Passing lists and individual list elements to methods. (Part 2 of 2.)

Lines 27–30 demonstrate passing a list element (**aList[3]**, which contains a number, recall that numbers are immutable) to a function. The program first prints the value of **aList[3]**, which is 8. Then, the program calls function **modifyElement** (lines 9–10) passing to parameter **element** the value 8. Function **modifyElement** multiplies **element** by 2. When the function terminates, the local variable **element** is destroyed. The value of the original element, **aList[3]**, in the list is not modified because the value of **aList[3]** is immutable. Thus, when control is returned to the main portion of the program, the unmodified value of **aList[3]** is printed.

Slicing creates a new sequence; therefore, when a program passes a slice to a function, the original sequence is not affected. Line 33 prints the slice **aList[2:4]** to the screen. Line 34 calls function **modifyList** and passes **aList[2:4]**. Line 35 prints the result of calling function **modifyList**—demonstrating that the original list was not modified.

Notice that function **modifyList** iterates through its list by accessing the elements using the square bracket operator. If the function contained the code

```
for item in aList:
   item *= 2
```

the list would remain unchanged, because the function would modify the value of local variable **item** and not the value stored at a particular index in the list.

5.9 Sorting and Searching Lists

Sorting data (i.e., placing the data into a particular order, such as ascending or descending) is a common computing application. For instance, a bank sorts checks by account number to prepare individual monthly bank statements. Telephone companies sort accounts by last names and, within that, by first names, to simplify the search for phone numbers. Almost all organizations sort data—in many cases, massive amounts of data. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science. In this section, we discuss how to sort a list using list method ***sort***.

Figure 5.17 sorts the values of the 10-element list **aList** (line 4) into ascending order. Lines 8–9 print the list items. Line 11 calls list method **sort**—this method sorts the ele-

ments of **aList** in ascending order. The remainder of the program prints the results of sorting the list.

Much research has been performed in the area of list-sorting algorithms, resulting in the design of many algorithms. Some of these algorithms are simple to express and program, but are inefficient. Other algorithms are complex and sophisticated, but provide increased performance. The exercises at the end of this chapter investigate a well-known sorting algorithm.

Performance Tip 5.1

Sometimes, the simplest algorithms perform poorly. Their virtue is that they are easy to write, test and debug. Sometimes complex algorithms are needed to realize maximum performance.

Often, programmers work with large amounts of data stored in lists. It might be necessary to determine whether a list contains a value that matches a certain *key value*. The process of locating a particular element value in a list is called *searching*.

The program in Fig. 5.18 searches a list for a value. Line 5 creates list **aList**, which contains the even numbers between 0 and 198, inclusive. Line 7 then retrieves the *search key* from the user and assigns the value to variable **searchKey**. Keyword **in** tests whether list **aList** contains the user-entered search key (line 9). If the list contains the value stored in variable **searchKey**, the expression (line 9) evaluates to true; otherwise, the expression evaluates to false.

```python
# Fig. 5.17: fig05_17.py
# Sorting a list.

aList = [ 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 ]

print "Data items in original order"

for item in aList:
   print item,

aList.sort()

print "\n\nData items after sorting"

for item in aList:
   print item,

print
```

```
Data items in original order
2 6 4 8 10 12 89 68 45 37

Data items after sorting
2 4 6 8 10 12 37 45 68 89
```

Fig. 5.17 Sorting a list.

```
1  # Fig. 5.18: fig05_18.py
2  # Searching a list for an integer.
3
4  # Create a list of even integers 0 to 198
5  aList = range( 0, 199, 2 )
6
7  searchKey = int( raw_input( "Enter integer search key: " ) )
8
9  if searchKey in aList:
10    print "Found at index:", aList.index( searchKey )
11 else:
12    print "Value not found"
```

```
Enter integer search key: 36
Found at index: 18
```

```
Enter integer search key: 37
Value not found
```

Fig. 5.18 Searching a list for an integer.

If the list contains the search key, line 10 invokes list method ***index*** to obtain the index of the search key. List method **index** takes a search key as a parameter, searches through the list and returns the index of the first list value that matches the search key. If the list does not contain any value that matches the search key, the program displays an error message. [*Note*: Figure 5.18 searches **aList** twice (lines 9–10), which, for large sequences, can result in poor performance. To improve performance, the program can use list method **index** and trap the exception that occurs if the argument is not in the list. We discuss exception-handling techniques in Chapter 12.]

As with sorting, a great deal of research has been devoted to the task of searching. In the exercises at the end of this chapter, we explore some of the more sophisticated ways of searching a list.

5.10 Multiple-Subscripted Sequences

Sequences can contain elements that are also sequences (i.e., lists and tuples). Such sequences have multiple subscripts. A common use of multiple-subscripted sequences is to represent *tables* of values consisting of information arranged in *rows* and *columns*. To identify a particular table element, we must specify two subscripts—by convention, the first identifies the element's row, the second the element's column.

Sequences that require two subscripts to identify a particular element are called *double-subscripted sequences* or *two-dimensional sequences*. Note that multiple-subscripted sequences can have more than two subscripts. Python does not support multiple-subscripted sequences directly, but allows programmers to specify single-subscripted tuples and lists whose elements are also single-subscripted tuples and lists, thus achieving the same effect. Figure 5.19 illustrates a double-subscripted sequence, **a**, containing three rows and four columns (i.e., a 3-by-4 sequence). In general, a sequence with *m* rows and *n* columns is called an *m-by-n sequence*.

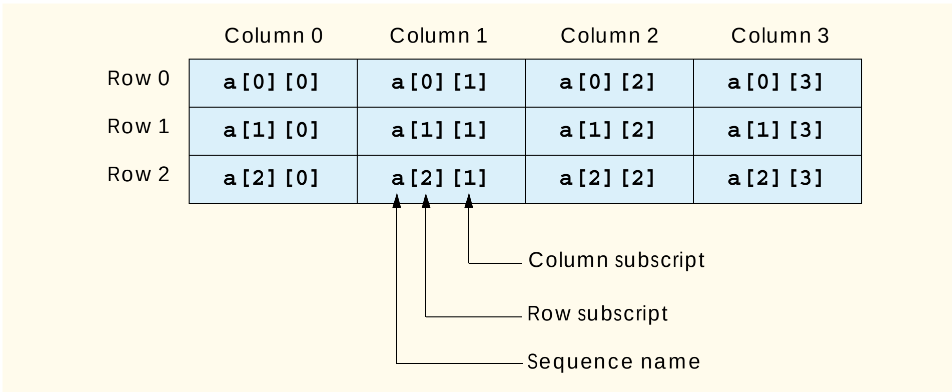


Fig. 5.19 Double-subscripted sequence with three rows and four columns.

Every element in sequence **a** is identified in Fig. 5.19 by an element name of the form **a[** ***i*** **][** ***j*** **]**; **a** is the name of the sequence, and ***i*** and ***j*** are the subscripts that uniquely identify the row and column of each element in **a**. Notice that the names of the elements in the first row all have 0 as the first subscript; the names of the elements in the fourth column all have 3 as the second subscript.

Multiple-subscripted sequences can be initialized during creation in much the same way as a single-subscripted sequence. A double-subscripted list with two rows and columns could be created with

```
b = [ [ 1, 2 ], [ 3, 4 ] ]
```

The values are grouped by row—the first row is the first element in the list, and the second row is the second element in the list. So, **1** and **2** initialize **b[0][0]** and **b[0][1]**, and **3** and **4** initialize **b[1][0]** and **b[1][1]**. Multiple-subscripted sequences are maintained as sequences of sequences. The statement

```
c = ( ( 1, 2 ), ( 3, 4, 5 ) )
```

creates a tuple **c** with row 0 containing two elements (**1** and **2**) and row 1 containing three elements (**3**, **4** and **5**). Python allows multiple-subscripted sequences to have rows of different lengths.

Figure 5.20 demonstrates creating and initializing double-subscripted sequences and using nested **for** structures to traverse the sequences (i.e., manipulate every element of the sequence).

```
# Fig. 5.20: fig05_20.py
# Making tables using lists of lists and tuples of tuples.

table1 = [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
table2 = ( ( 1, 2 ), ( 3, ), ( 4, 5, 6 ) )

```

Fig. 5.20 Tables created using lists of lists and tuples of tuples. (Part 1 of 2.)

```
7   print "Values in table1 by row are"
8
9   for row in table1:
10
11     for item in row:
12        print item,
13
14     print
15
16  print "\nValues in table2 by row are"
17
18  for row in table2:
19
20     for item in row:
21        print item,
22
23     print
```

```
Values in table1 by row are
1 2 3
4 5 6

Values in table2 by row are
1 2
3
4 5 6
```

Fig. 5.20 Tables created using lists of lists and tuples of tuples. (Part 2 of 2.)

The program declares two sequences. Line 4 creates the multiple-subscript list **table1** and provides six values in two sublists (i.e., two lists-within-lists). The first sublist (row) of the sequence contains the values 1, 2 and 3; the second sublist contains the values 4, 5 and 6.

Line 5 creates multiple-subscript tuple **table2** and provides six values in three subtuples (i.e., tuples-within-tuples). The first subtuple (row) contains two elements with values 1 and 2, respectively. The second subtuple contains one element with value 3. The third subtuple contains three elements with values 4, 5 and 6. Lines 9–14 use a nested **for** structure to output the rows of list **table1**. The outer **for** structure iterates over the rows in the list. The inner **for** structure iterates over each column in the row. The remainder of the program prints the values for variable **table2** in a similar manner.

The program in Fig. 5.20 demonstrates one case in a which a **for** structure is useful for manipulating a multiple-subscripted sequence. Many other common sequence manipulations use **for** repetition structures. For example, the following **for** structure sets all the elements in the third row of sequence **a** in Fig. 5.19 to 0:

```
for column in range( len( a[ 2 ] ) ):
   a[ 2 ][ column ] = 0
```

We specified the *third* row; thus, the first subscript is always 2 (0 is the first row and 1 is the second row). The **for** structure varies only the second subscript (i.e., the column subscript). The preceding **for** structure is equivalent to the assignment statements

```
a[ 2 ][ 0 ] = 0
a[ 2 ][ 1 ] = 0
a[ 2 ][ 2 ] = 0
a[ 2 ][ 3 ] = 0
```

The following nested **`for`** structure determines the total of all the elements in sequence **`a`**:

```
total = 0

for row in a:
   for column in row:
      total += column
```

The **`for`** structure totals the elements of the sequence one row at a time. The outer **`for`** structure iterates over the rows in the table so that the elements of each row may be totaled by the inner **`for`** structure. The **`total`** is displayed when the nested **`for`** structure terminates.

The program in Fig. 5.21 performs several other common sequence manipulations on the 3-by-4 list **`grades`**. Each row of the list represents a student, and each column represents a grade on one of the four exams the students took during the semester. The list manipulations are performed by four functions. Function **`printGrades`** (lines 5–25) prints the data stored in list **`grades`** in a tabular format. Function **`minimum`** (lines 28–38) determines the lowest grade of any student for the semester. Function **`maximum`** (lines 41–51) determines the highest grade of any student for the semester. Function **`average`** (lines 54–60) determines a particular student's semester average. Notice that line 55 initializes **`total`** to **`0.0`**, so the function returns a floating-point value.

```
1  # Fig. 5.21: fig05_21.py
2  # Double-subscripted list example.
3
4
5  def printGrades( grades ):
6     students = len( grades )      # number of students
7     exams = len( grades[ 0 ] )    # number of exams
8
9     # print table headers
10    print "The list is:"
11    print "           ",
12
13    for i in range( exams ):
14       print "[%d]" % i,
15
16    print
17
18    # print scores, by row
19    for i in range( students ):
20       print "grades[%d]  " % i,
21
22       for j in range( exams ):
23          print grades[ i ][ j ], "",
24
25       print
```

Fig. 5.21 Double-scripted tuples. (Part 1 of 3.)

```python


def minimum( grades ):
   lowScore = 100

   for studentExams in grades:      # loop over students

      for score in studentExams:    # loop over scores

         if score < lowScore:
            lowScore = score

   return lowScore


def maximum( grades ):
   highScore = 0

   for studentExams in grades:      # loop over students

      for score in studentExams:    # loop over scores

         if score > highScore:
            highScore = score

   return highScore


def average( setOfGrades ):
   total = 0.0

   for grade in setOfGrades:        # loop over student's scores
      total += grade

   return total / len( setOfGrades )


# main program
grades = [ [ 77, 68, 86, 73 ],
           [ 96, 87, 89, 81 ],
           [ 70, 90, 86, 81 ] ]

printGrades( grades )
print "\n\nLowest grade:", minimum( grades )
print "Highest grade:", maximum( grades )
print "\n"

# print average for each student
for i in range( len( grades ) ):
   print "Average for student", i, "is", average( grades[ i ] )
```

Fig. 5.21 Double-scripted tuples. (Part 2 of 3.)

```
The list is:
             [0]  [1]  [2]  [3]
grades[0]    77   68   86   73
grades[1]    96   87   89   81
grades[2]    70   90   86   81


Lowest grade: 68
Highest grade: 96


Average for student 0 is 76.0
Average for student 1 is 88.25
Average for student 2 is 81.75
```

Fig. 5.21 Double-scripted tuples. (Part 3 of 3.)

Function **printGrades** uses the list **grades** and variables **students** (number of rows in the list) and **exams** (number of columns in the list). The function loops through list **grades**, using nested **for** structures to print out the grades in tabular format. The outer **for** structure (lines 19–25) iterates over **i** (i.e., the row subscript), the inner **for** structure (lines 22–23) over **j** (i.e., the column subscript).

Functions **minimum** and **maximum** loop through list **grades**, using nested **for** structures. Function **minimum** compares each grade to variable **lowScore**. If a grade is less than **lowScore**, **lowScore** is set to that grade (line 36). When execution of the nested structure is complete, **lowScore** contains the smallest grade in the double-subscripted list. Function **maximum** works similarly to function **minimum**.

Function **average** takes one argument—a single-subscripted list of test results for a particular student. When line 75 invokes **average**, the argument is **grades[i]**, which specifies that a particular row of the double-subscripted list **grades** is to be passed to **average**. For example, the argument **grades[1]** represents the four values (a single-subscripted list of grades) stored in the second row of the double-subscripted list **grades**. Remember that, in Python, a double-subscripted list is a list with elements that are single-subscripted lists. Function **average** calculates the sum of the list elements, divides the total by the number of test results and returns the floating-point result.

In the above example, we demonstrated how to use double-subscripted lists. However, when we need to compute pure numerical problems (i.e., multi-dimensional arrays), the basic Python language cannot handle them efficiently. In this case, a package called **NumPy** should be used. The **NumPy** (numerical python) package contains modules that handle arrays, and it provides multi-dimensional array objects for efficient computation. For more information on **NumPy**, visit **sourceforge.net/projects/numpy**.

Chapters 2–5 introduced the basic-programming techniques of Python. In Chapter 6, Introduction to the Common Gateway Interface (CGI), we will use these techniques to design Web-based applications. In Chapters 7–9, we will introduce object-oriented programming techniques that will allow us to build complex applications in the latter half of the book.

SUMMARY

- Data structures hold and organize information (data).
- Sequences, often called arrays in other languages, are data structures that store related data items. Python supports three basic sequence data types: a string, a list and a tuple.
- A sequence element may be referenced by writing the sequence name followed by the element's position number in square brackets (**[]**). The first element in a sequence is the zeroth element.
- Sequences can be accessed from the end of the sequence by using negative subscripts.
- The position number more formally is called a subscript (or an index), which must be an integer or an integer expression. If a program uses an integer expression as a subscript, Python evaluates the expression to determine the location of the subscript.
- Some types of sequences are immutable—the sequence cannot be altered (e.g., by changing the value of one of its elements). Python strings and tuples are immutable sequences.
- Some sequences are mutable—the sequence can be altered. Python lists are mutable sequences.
- The length of the sequence is determined by the function call **len(** *sequence* **)**.
- To create an empty string, use the empty quotes (i.e., **""**, **''**,**""" """** or **''' '''**)
- To create an empty list, use empty square brackets (i.e., **[]**). To create a list that contains a sequence of values, separate the values with commas, and place the values inside square brackets.
- To create an empty tuple, use the empty parentheses (i.e., **()**). To create a tuple that contains a sequence of values, simply separate the values with commas. Tuples also can be created by surrounding the tuple values with parentheses; however, the parentheses are optional.
- Creating a tuple is sometimes referred to as packing a tuple.
- When creating a one-element tuple—called a singleton—write the value, followed by a comma (**,**).
- In practice, Python programmers distinguish between tuples and lists to represent different kinds of sequences, based on the context of the program.
- Although lists are not restricted to homogeneous data types, Python programmers typically use lists to store sequences of homogeneous values—values of the same data type. In general, a program uses a list to store homogeneous values for the purpose of looping over these values and performing the same operation on each value. Usually, the length of the list is not predetermined and may vary over the course of the program.
- The **+=** augmented assignment statement can insert a value in a list. When the value to the left of the **+=** symbol is a sequence, the value to the right of the symbol must be a sequence also.
- The **for**/**in** structure iterates over a sequence. The **for** structure starts with the first element in the sequence, assigns the value of the first element to the control variable and executes the body of the **for** structure. Then, the **for** structure proceeds to the next element in the sequence and performs the same operations.
- If a program attempts to access a nonexistent index, the program exits and displays an "out-of-range" error message. This error can be caught as an exception.
- Tuples store sequences of heterogeneous data. Each data piece in a tuple represents a part of the total information represented by the tuple. Usually, the length of the tuple is predetermined and does not change over the course of a program's execution. A program usually does not iterate over a sequence, but accesses the parts of the tuple the program needs to perform its task.
- If a program attempts to modify a tuple, the program exits and displays an error message.
- Sequences can be unpacked—the values stored in the sequence are assigned to various identifiers. Unpacking is a useful programming shortcut for assigning values to multiple variables in a single statement.

- When unpacking a sequence, the number of variable names to the left of the **=** symbol must equal the number of elements in the sequence to the right of the symbol.
- Python provides the slicing capability to obtain contiguous regions of a sequence.
- To obtain a slice of the *i*th element through the *j*th element, inclusive, use the expression *sequence* **[i:j + 1]**.
- The dictionary is a mapping construct that consists of key-value pairs. Dictionaries (called hashes or associative arrays in other languages), can be thought of as unordered collections of values where each value is accessed through its corresponding key.
- To create an empty dictionary, use empty curly braces (i.e., **{}**).
- To create a dictionary with values, use a comma-separated sequence of key-value pairs, inside curly braces. Each key-value pair is of the form *key* : *value*.
- Python dictionary keys must be immutable values, like strings, numbers or tuples, whose elements are immutable. Dictionary values can be of any Python data type.
- Dictionary values are accessed with the expression *dictionaryName*[*key*].
- To insert a new key-value pair in a dictionary, use the statement *dictionaryName*[*key*] = *value*.
- The statement *dictionaryName* **[** *key* **] =** *value* modifies the value associated with key, if the dictionary already contains that key. Otherwise, the statement inserts the key-value pair into the dictionary.
- Accessing a non-existent dictionary key causes the program to exit and to display a "key error" message.
- A method performs the behaviors (tasks) of an object.
- To invoke an object's method, specify the name of the object, followed by the dot (**.**) access operator, followed by the method invocation.
- List method **append** adds an items to the end of a list.
- List method **count** takes a value as an argument and returns the number of elements in the list that have that value. If the list contains no elements with the specified value, method **count** returns 0.
- Dictionary method **items** returns a list of tuples, where each tuple contains a key-value pair. Dictionary method **keys** returns an unordered list of the dictionary's keys. Dictionary method **values** returns an unordered list of the dictionary's values.
- Dictionary method **copy** returns a new dictionary that is a shallow copy of the original dictionary. In a shallow copy, the elements in the new dictionary are references to the elements in the original dictionary.
- If the programmer wants to create a copy—called a deep copy—that is independent of the original dictionary, Python provides module **copy**. Function **copy.deepcopy** returns a deep copy of it argument.
- In many programming languages, the two ways to pass arguments to functions are pass-by-value and pass-by-reference (also called pass-by-value and pass-by-reference).
- When an argument is passed by value, a copy of the argument's value is made and passed to the called function.
- With by reference, the caller allows the called function to access the caller's data directly and to modify that data.
- Unlike many other languages, Python does not allow programmers to choose between pass-by-value and pass-by-reference to pass arguments. Python arguments are always passed by object reference—the function receives references to the values passed as arguments. In practice, pass-by-object-reference can be thought of as a combination of pass-by-value and pass-by-reference.

- If a function receives a reference to a mutable object (e.g., a dictionary or a list), the function can modify the original value of the object. It is as if the object had been passed by reference.
- If a function receives a reference to an immutable object (e.g., a number, a string or a tuple whose elements are immutable values), the function cannot modify the original object directly. It is as if the object had been passed by value.
- To pass a list argument to a function, specify the name of the list without square brackets.
- Although entire lists can be changed by a function, individual list elements that are numeric and immutable sequence data types cannot be changed. To pass a list element to a function, use the subscripted name of the list element as an argument in the function call.
- Slicing creates a new sequence; therefore, when a program passes a slice to a function, the original sequence is not affected.
- Sorting data is the process of placing data into a particular order.
- By default, list method **`sort`** sorts the elements of a list in ascending order.
- Some sorting algorithms are simple to express and program, but are inefficient. Other algorithms are complex and sophisticated, but provide increased performance.
- Often, programmers work with large amounts of data stored in lists. It might be necessary to determine whether a list contains a value that matches a certain key value. The process of locating a particular element value in a list is called searching.
- Keyword **`in`** tests whether a sequence contains a particular value.
- List method **`index`** takes a search key as a parameter, searches through the list and returns the index of the first list value that matches the search key. If the list does not contain any value that matches the search key, the program displays an error message.
- Sequences can contain elements that are also sequences. Such sequences have multiple subscripts. A common use of multiple-subscripted sequences is to represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, we must specify two subscripts—by convention, the first identifies the element's row, the second identifies the element's column.
- Sequences that require two subscripts to identify a particular element are called double-subscripted sequences or two-dimensional sequences.
- Python does not support multiple-subscripted sequences directly, but allows programmers to specify single-subscripted tuples and lists whose elements are also single-subscripted tuples and lists, thus achieving the same effect.
- A sequence with *m* rows and *n* columns is called an m-by-n sequence. It is more commonly know as two-dimensional sequence.
- The name of every element in a multiple-subscripted sequence is of the form **`a[ i ][ j ]`**, where **`a`** is the name of the sequence, and ***`i`*** and ***`j`*** are the subscripts that uniquely identify the row and column of each element in the sequence.
- To compute pure numerical problems (i.e., multi-dimensional arrays), use package **`NumPy`** (numerical Python). This package contains modules that handle arrays and provides multi-dimensional array objects for efficient computation.

TERMINOLOGY

`append` method of list
array
associative array
bracket operator (**`[]`**)
`clear` method of dictionary
column

comma (**,**)
copy method of dictionary
count list method
data structure
deep copy of a dictionary
dictionary
dictionary method
double-subscripted sequence
dot access operator (**.**)
element
empty curly braces **{}**
empty dictionary
empty list
empty parentheses **()**
empty quotes
empty square brackets **[]**
empty string
empty tuple
for structure
get method of dictionary
hash
has_key method of dictionary
heterogeneous data (in tuples)
histogram
homogeneous data (in lists)
immutable sequence
in keyword
index
in-place sorting
index method of list
items method of dictionary
iteritems method of dictionary
iterkeys method of dictionary
itervalues method of dictionary
keys method of dictionary
key value
key-value pair
length (sequence)
list
list method
m-by-*n* sequence
mapping construct
method
method invocation
multiple-subscripted sequence
mutable sequence
name (sequence)
NumPy package (numerical Python)
one-element tuple (singleton)
out-of-range error message
packed
packing a tuple
pass-by-object-reference
pass-by-reference
pass-by-value
popitem method of dictionary
position number
row
search
search key
sequence
sequence slicing
sequence unpacking
setdefault method of dictionary
shallow copy of a dictionary
singleton
slice a sequence
slicing operator (**[:]**)
sort
sort list method
subscript
table
tuple
two-dimensional sequence
update method of dictionary
unpacked sequence
value (sequence)
values dictionary method
zeroth element

SELF-REVIEW EXERCISES

5.1 Fill in the blanks in each of the following statements:

a) __________ are “associative arrays” that consist of __________ pairs.
b) The last element in a sequence can always be accessed with subscript __________.
c) Statement __________ creates a singleton **aTuple**.
d) Function __________ returns the length of a sequence.
e) Selecting a portion of a sequence with the operator **[:]** is called __________.
f) Dictionary method __________ returns a list of key-value pairs.

g) When an argument is passed __________, a copy of the argument's value is made and passed to the called method.
h) Use the expression __________ to obtain the *i*th element through the *j*th element of list **sequence**, inclusive.
i) A sequence with *m* rows and *n* columns is called an __________.
j) List method __________ returns the number of times a specified element occurs in a list.

5.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) A sequence begins at subscript 1.
b) Strings and tuples are mutable sequences.
c) Each key-value pair in a dictionary has the form *key* **:** *value*.
d) Using a tuple as a dictionary key is an error.
e) Dictionary values are accessed with the dot operator.
f) Method **insert** adds one element to the end of a list.
g) The **+=** statement appends items into lists.
h) List method **sort** sorts the elements of a list in place.
i) If list method **search** finds a list value that matches the search key, it returns the subscript of the list value.
j) Unlike other languages, Python does not allow the programmer to choose whether to pass each argument pass-by-value or pass-by-reference.

ANSWERS TO SELF-REVIEW EXERCISES

5.1 a) Dictionaries, key-value. b) -1. c) **aTuple = 1,**. d) **len**. e) slicing. f) **items**. g) pass-by-value. h) **sequence[i:j + 1]**. i) m-by-n sequence. j) **count**.

5.2 a) False. The first element in every sequence has subscript 0. b) False. Strings and tuples are immutable sequences—their values cannot be altered. c) True. d) False. Dictionary keys must be immutable data types, such as tuples. e) False. Dictionary values are accessed with the expression *dictionaryName*[*key*]. f) False. Method **append** adds one element to the end of a list. g) True. h) True. i) False. If list method **index** finds a list value that matches the search key, it returns the subscript of the list value. j) True.

EXERCISES

5.3 Use a list to solve the following problem: Read in 20 numbers. As each number is read, print it only if it is not a duplicate of a number already read.

5.4 Use a list of lists to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains:
a) The salesperson number.
b) The product number.
c) The number of that product sold that day.

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write a program that will read all this information for last month's sales and summarize the total sales by salesperson by product. All totals should be stored in list **sales**. After processing all the information for last month, display the results in tabular format, with each of the columns representing a particular salesperson and each of the rows representing a particular product. Cross-total each row to get the total sales of each product for last month; cross-total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross-totals to the right of the totaled rows and at the bottom of the totaled columns.

5.5 (*The Sieve of Eratosthenes*) A prime integer is any integer greater than 1 that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a) Create a list with all elements initialized to 1 (true). List elements with prime subscripts will remain 1. All other list elements will eventually be set to zero.
b) Starting with list element 2, every time a list element is found whose value is 1, loop through the remainder of the list and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For list subscript 2, all elements beyond 2 in the list that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, etc.); for list subscript 3, all elements beyond 3 in the list that are multiples of 3 will be set to zero (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the list elements that are still set to 1 indicate that the subscript is a prime number. These subscripts can then be printed. Write a program that uses a list of 1000 elements to determine and print the prime numbers between 2 and 999. Ignore element 0 of the list.

5.6 (*Bubble Sort*) Sorting data (i.e. placing data into some particular order, such as ascending or descending) is one of the most important computing applications. Python lists provide a **`sort`** method. In this exercise, readers implement their own sorting function, using the bubble-sort method. In the bubble sort (or *sinking* sort), the smaller values gradually "bubble" their way upward to the top of the list like air bubbles rising in water, while the larger values sink to the bottom of the list. The process that compares each adjacent pair of elements in a list in turn and swaps the elements if the second element is less than the first element is called a pass. The technique makes several passes through the list. On each pass, successive pairs of elements are compared. If a pair is in increasing order, bubble sort leaves the values as they are. If a pair is in decreasing order, their values are swapped in the list. After the first pass, the largest value is guaranteed to sink to the highest index of a list. After the second pass, the second largest value is guaranteed to sink to the second highest index of a list, and so on. Write a program that uses function **`bubbleSort`** to sort the items in a list.

5.7 (*Binary Search*) When a list is sorted, a high-speed binary search technique can find items in the list quickly. The binary search algorithm eliminates from consideration one-half of the elements in the list being searched after each comparison. The algorithm locates the middle element of the list and compares it with the search key. If they are equal, the search key is found, and the subscript of that element is returned. Otherwise, the problem is reduced to searching one half of the list. If the search key is less than the middle element of the list, the first half of the list is searched. If the search key is not the middle element in the specified piece of the original list, the algorithm is repeated on one-quarter of the original list. The search continues until the search key is equal to the middle element of the smaller list or until the smaller list consists of one element that is not equal to the search key (i.e. the search key is not found.)

Even in a worst-case scenario, searching a list of 1024 elements will take only 10 comparisons during a binary search. Repeatedly dividing 1024 by 2 (because after each comparison we are able to eliminate from the consideration half the list) yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 (210) is divided by 2 only ten times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary-search algorithm. A list of 1,048,576 (2^{20}) elements takes a maximum of 20 comparisons to find the key. A list of one billion elements takes a maximum of 30 comparisons to find the key. The maximum number of comparisons needed for the binary search of any sorted list can be determined by finding the first power of 2 greater than or equal to the number of elements in the list.

Write a program that implements function **`binarySearch`**, which takes a sorted list and a search key as arguments. The function should return the index of the list value that matches the search key (or -1, if the search key is not found).

5.8 Create a dictionary of 20 random values in the range 1–99. Determine whether there are any duplicate values in the dictionary. (*Hint*: you many want to sort the list first.)

6

Introduction to the Common Gateway Interface (CGI)

Objectives

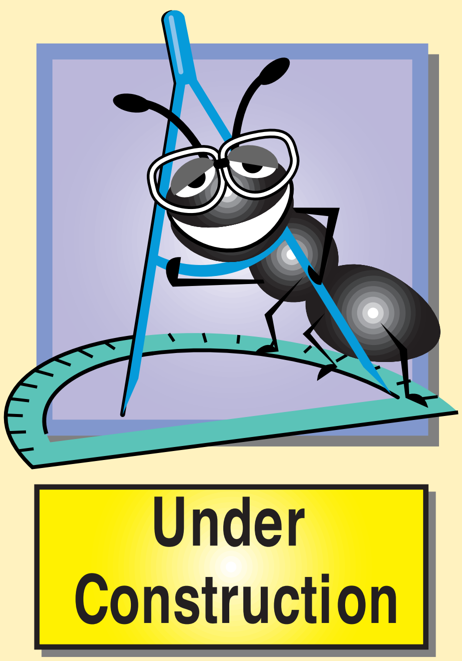
- To understand the Common Gateway Interface (CGI) protocol.
- To understand the Hypertext Transfer Protocol (HTTP).
- To implement CGI scripts.
- To use XHTML forms to send information to CGI scripts.
- To understand and parse query strings.
- To use module **`cgi`** to process information from XHTML forms.

This is the common air that bathes the globe.
Walt Whitman

The longest part of the journey is said to be the passing of the gate.
Marcus Terentius Varro

Railway termini...are our gates to the glorious and unknown. Through them we pass out into adventure and sunshine, to them, alas! we return.
E. M. Forster

There comes a time in a man's life when to get where he has to go—if there are no doors or windows—he walks through a wall.
Bernard Malamud



Outline

6.1 Introduction

The *Common Gateway Interface (CGI)* describes a set of protocols through which applications (commonly called *CGI programs* or *CGI scripts*) interact with Web servers and indirectly with Web browsers (e.g., client applications). A Web server is a specialized software application that responds to client application requests by providing resources (e.g. Web pages). CGI protocols often generate Web content dynamically. A Web page is dynamic if a program on the Web server generates that page's content each time a user requests the page. For example, a form in a Web page could request that a user enter a zip code. When the user types and submits the zip code, the Web server can use a CGI program to create a page that displays information about the weather in that client's region. In contrast, *static* Web page content never changes unless the Web developers edit the document.

CGI is "common" because it is not specific to any operating system (e.g., Linux or Windows), to any programming language or to any Web server software. CGI can be used with virtually any programming or scripting language, such as C, Perl and Python. In this chapter, we explain how Web clients and servers interact. We introduce the basics of CGI and use Python to write CGI scripts.

The CGI protocol was developed in 1993 by the *National Center for Supercomputing Applications* (*NCSA*—**`www.ncsa.uiuc.edu`**), for use with its *HTTPd Web server*. NCSA developed CGI to be a simple tool to produce dynamic Web content. The simplicity of CGI resulted in its widespread use and in its adoption as an unofficial worldwide protocol. CGI was quickly incorporated into additional Web servers, such as Microsoft *Internet Information Services (IIS)* and Apache (**`www.apache.org`**).

6.2 Client and Web Server Interaction

In this section, we discuss the interactions between a Web server and a client application. A Web page, in its simplest form, is either a *Hypertext Markup Language* (*HTML)* document or an *Extensible Hypertext Markup Language (XHTML)* document. (In this chapter, we use XHTML.) An XHTML document is a plain-text file that contains markup, or *tags*, which describe how the document should be displayed by a Web browser. For example, the XHTML markup

```
<title>My Web Page</title>
```

indicates that the text between the opening **`<title>`** tag and the closing **`</title>`** tag is the Web page's title. The browser renders the text between these tags in a specific manner.

XHTML requires *syntactically* correct documents—markup must follow specific rules. For example, XHTML tags must be in all lowercase letters and all opening tags must have corresponding closing tags. We discuss XHTML in detail in Appendix I and Appendix J.

Each Web page has a unique *Uniform Resource Locator* (*URL*) associated with it—an address of sorts. The URL contains information that directs a browser to the resource (most often a Web page) the user wishes to access. For example, consider the URL

```
http://www.deitel.com/books/downloads.html
```

The first part of the address, **`http://`**, indicates that the resource is to be obtained using the *Hypertext Transfer Protocol (HTTP)*. During this interaction, the Web server and the client communicate using the platform-independent HTTP, a protocol for transferring requests and files over the Internet (e.g., between Web servers and Web browsers). Section 6.2.3 discusses HTTP.

The next section of the URL—**`www.deitel.com`**—is the *hostname* of the server, which is the name of the server computer, the *host*, on which the resource resides. A *domain name system (DNS*) server translates the hostname (**`www.deitel.com`**) into an *Internet Protocol* (*IP) address* (e.g., **`207.60.134.230`**) that identifies the server computer (just as a telephone number uniquely identifies a particular phone line). This translation operation is a *DNS lookup*. A DNS server maintains a database of hostnames and their corresponding IP addresses.

The remainder of the URL specifies the requested resource—**`/books/downloads.html`**. This portion of the URL specifies both the name of the resource (**`downloads.html`**—an HTML/XHTML document) and its path (**`/books`**). The Web server maps the URL to a file (or other resource, such as a CGI program) on the server, or to another resource on the server's network. The Web server then returns the requested document to the client. The path represents a directory in the Web server's file system. It also is possible that the resource is created dynamically and does not reside anywhere on the server computer. In this case, the URL uses the hostname to locate the correct server, and the server uses the path and resource information to locate (or create) the resource to respond to the client's request. As we will see, URLs also can provide input to a CGI program residing on a server.

6.2.1 System Architecture

A Web server often is part of a *multi-tier application*, sometimes referred to as an *n*-tier application. Multi-tier applications divide functionality into separate *tiers* (i.e., logical

groupings of functionality). Tiers can be located on a single computer or on multiple computers. Figure 6.1 presents the basic structure of a three-tier application.

The *information tier* (also called the *data tier* or the *bottom tier*) maintains data for the application. This tier typically stores data in a *relational database management system (RDBMS)*. We discuss relational database management systems in further detail in Chapter 17, Database Application Programming Interface (DB-API). For example, a retail store may have a database for product information, such as descriptions, prices and quantities in stock. The same database also may contain customer information, such as user names, billing addresses and credit-card numbers.

The *middle tier* implements *business logic* and *presentation logic* to control interactions between application clients and application data. The middle tier acts as an intermediary between data in the information tier and the application clients. The middle-tier *controller logic* processes client requests from the client tier (e.g., a request to view a product catalog) and retrieves data from the database. The middle-tier presentation logic then processes data from the information tier and presents the content to the client.

Business logic in the middle tier enforces *business rule*s and ensures that data are reliable before updating the database or presenting data to a client. Business rules dictate how clients can and cannot access application data and how applications process data.

The middle tier also implements the application's presentation logic. Web applications typically present information to clients as XHTML documents (older applications present information as HTML). Many Web applications present information to wireless clients as Wireless Markup Language (WML) documents. We discuss WML in detail in Chapter 23, Case Study: Online Bookstore.

The *client tier*, or *top tier*, is the application's user interface. Users interact with the application through the user interface. This causes the client to interact with the middle tier to make requests and to retrieve data from the information tier. The client then displays to the user the data retrieved from the middle tier.

6.2.2 Accessing Web Servers

To request documents from Web servers, users must know the machine names (called hostnames) on which Web server software resides. Users can request documents from *local Web servers* (i.e, those that reside on users' machines) or *remote Web servers* (i.e., those that reside on different machines).

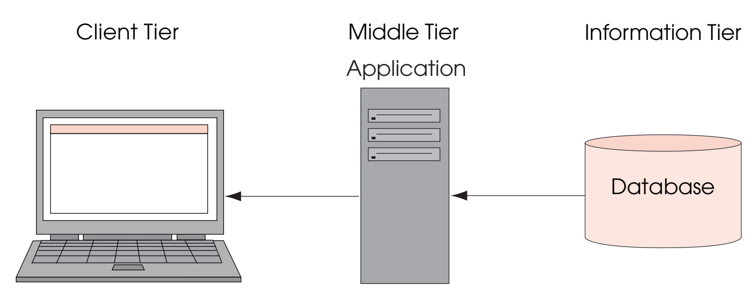


Fig. 6.1 Three-tier application model.

We can request document from local Web servers through the machine name or through ***localhost***—a hostname that references the local machine. We use **localhost** in this book. To determine the machine name in Windows 98, right-click **Network Neighborhood**, and select **Properties** from the context menu to display the **Network** dialog. In the **Network** dialog, click the **Identification** tab. The computer name displays in the **Computer name:** field. Click **Cancel** to close the **Network** dialog. In Windows 2000, right click **My Network Places** and select **Properties** from the context menu to display the **Network and Dialup Connections** explorer. In the explorer, click **Network Identification**. The **Full Computer Name:** field in the **System Properties** window displays the computer name. To determine the machine name on most Linux machines, simply type the command **hostname** at a shell prompt.

A client also can access a server by specifying the server's domain name or IP address (e.g., in a Web browser's **Address** field). A domain name represents a group of hosts on the Internet; it combines with a hostname (such as **www**—a common hostname for Web servers) and a *top-level domain (TLD)* to form a *fully qualified hostname*, which provides a user-friendly way to identify a site on the Internet. In a fully qualified hostname, the TLD often describes the type of organization that owns the domain name. For example, the **com** TLD usually refers to a commercial business, whereas the **org** TLD usually refers to a non-profit organization. In addition, each country has its own TLD, such as **cn** for China, **et** for Ethiopia, **om** for Oman and **us** for the United States.

6.2.3 HTTP Transactions

Before exploring how CGI operates, it is necessary to have a basic understanding of networking and the World Wide Web. In this section, we discuss the technical aspects of how a browser interacts with a Web server to display a Web page and we examine the Hypertext Transfer Protocol (HTTP). We also explore HTTP's components that enable clients and servers to interact and exchange information uniformly and predictably.

An HTTP request often posts data to a *server-side form handler* that processes the data. For example, when a user participates in a Web-based survey, the Web server receives the information specified in the XHTML form as part of the request.

When a user enters a URL, the client has to request that resource. The two most common *HTTP request types* (also known as *request methods*) are *get* and *post*. These request types retrieve resources from a Web server and send client form data to a Web server. A *get* request sends form content as part of the URL. For example, in the URL

www.*somesite***.com/search?query=***value*

the information following the **?** (**query=***value*) indicates the user-specified input. For example, if the user performs a search on "Massachusetts," the last part of the URL would be **?query=Massachusetts**. Most Web servers limit *get* request query strings to 1024 characters. If the query string exceeds this limit, the *post* request must be used. The data sent in a *post* request is not part of the URL and cannot be seen by the user. Forms that contain many fields are submitted most often by *post* requests. Sensitive form fields, such as passwords, usually are sent using this request type.

To make the request, the browser sends an HTTP request message to the server (step 1, Fig. 6.2). HTTP has two request types, *get* and *post*. The *get* request (in its simplest form) follows the format: **GET /books/downloads.html HTTP/1.1**. The word ***GET*** is an

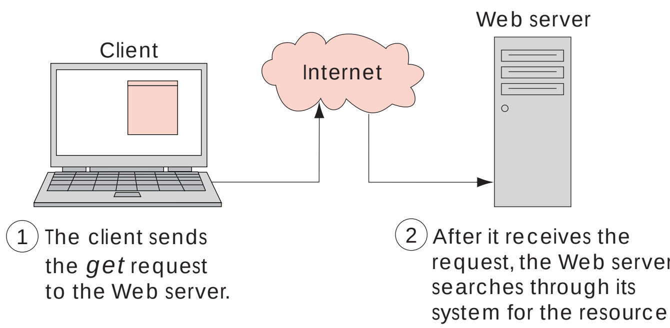


Fig. 6.2 Client interacting with server and Web server. Step 1: The request, **`GET /books/downloads.html HTTP/1.1`**.

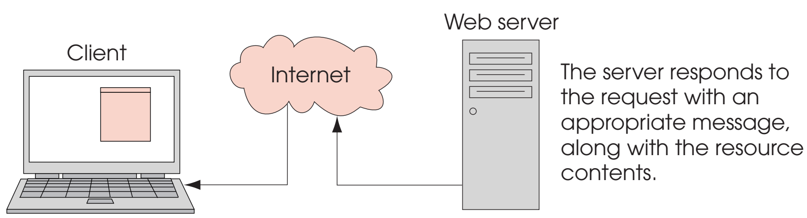


Fig. 6.2 Client interacting with server and Web server. Step 2: The HTTP response, **`HTTP/1.1 200 OK`**.

HTTP method indicating that the client is requesting a resource. The next part of the request provides the name (**`downloads.html`**) and path (**`/books/`**) of the resource (an HTML/XHTML document). The final part of the request provides the protocol's name and version number (**`HTTP/1.1`**).

Servers that understand HTTP version 1.1 translate this request and respond (step 2, Fig. 6.2). The server responds with a line indicating the HTTP version, followed by a status code that consists of a numeric code and phrase describing the status of the transaction. For example,

```
HTTP/1.1 200 OK
```

indicates success, while

```
HTTP/1.1 404 Not found
```

informs the client that the requested resource was not found on the server in the location specified by the URL.

Browsers often *cache* (save on a local disk) Web pages for quick reloading, to reduce the amount of data that the browser needs to download. However, browsers typically do not cache server responses to *post* requests, because subsequent *post* requests may not contain the same information. For example, several users who participate in a Web-based survey

may request the same Web page. Each user's response changes the overall results of the survey, thus the data on the Web server is changed.

On the other hand, Web browsers cache server responses to *get* requests. With a Web-based search engine, a *get* request normally supplies the search engine with search criteria specified in an XHTML form. The search engine then performs the search and returns the results as a Web page. These pages are cached in the event that the user performs the same search again.

The server normally sends one or more *HTTP headers*, which provide additional information about the data sent in response to the request. In this case, the server is sending an HTML/XHTML text document, so the HTTP header reads

```
Content-type: text/html
```

This information is known as the *MIME (Multipurpose Internet Mail Extensions) type* of the content. MIME is an Internet standard that specifies how messages should be formatted, and clients use the content type to determine how to represent the content to the user. Each type of data sent has a MIME type associated with it that helps the browser determine how to process the data it receives. For example, the MIME type **`text/plain`** indicates that the data is text that should be displayed without attempting to interpret any of the content as HTML or XHTML markup. Similarly, the MIME type **`image/gif`** indicates that the content is a *GIF (Graphics Interchange Format)* image. When this MIME type is received by the browser, it attempts to display the image. For more information on MIME, visit

```
www.nacs.uci.edu/indiv/ehood/MIME/MIME.html
```

The header (or set of headers) is followed by a blank line (a carriage return, line feed or combination of both) which indicates to the client that the server is finished sending HTTP headers. The server then sends the text in the requested HTML/XHTML document (**`downloads.html`**). The connection terminates when the transfer of the resource completes. The client-side browser interprets the text it receives and displays (or renders) the results.

This section examined how a simple HTTP transaction is performed between a Web-browser application on the client side (e.g., Microsoft Internet Explorer or Netscape Communicator) and a Web-server application on the server side (e.g., Apache or IIS). Next, we introduce CGI programming.

6.3 Simple CGI Script

Two types of scripting are used in Web-based applications: *server-side* and *client-side*. CGI scripts are an example of server-side scripts because they run on the server. Programmers have greater control over Web page content when using server-side scripts, because server-side scripts can manipulate databases and other server resources. An example of client-side scripting is JavaScript. Client-side scripts can access the browser's features, manipulate browser documents, validate user input and much more.

Scripts executed on the server usually generate custom responses for clients. For example, a client might connect to an airline's Web server and request a list of all flights from Boston to San Antonio between September 19th and November 5th. The server queries the database, dynamically generates XHTML content containing the flight list and sends the XHTML to the client. This technology allows clients to obtain the most current flight information from the database by connecting to an airline's Web server.

Server-side scripting languages have a wider range of programmatic capabilities than their client-side equivalents. For example, server-side scripts can access the server's file directory structure, whereas client-side scripts cannot access the client's file directory structure.

Server-side scripts also have access to server-side software that extends server functionality. These pieces of software are called *COM components* for Microsoft Web servers and *modules* for Apache Web servers. Components and modules range from programming language support to counting the number of times a Web page has been visited (known as the number of *hits*).

Software Engineering Observation 6.1

Server-side scripts are not visible to the client; only the content the server delivers is visible to the client.

As long as a file on the server remains unchanged, its associated URL will display the same content in clients' browsers each time the file is accessed. For the content in the file to change (e.g., to include new links or the latest company news), someone must alter the file manually (probably with a text editor or Web-page design software) then load the changed file back onto the server.

Manually changing Web pages is not feasible for those who want to create interesting and dynamic Web pages. For example, if you want your Web page always to display the current date or weather, the page would require continuous updating.

The examples in this chapter rely heavily upon XHTML and *Cascading Style Sheets (CSS)*. CSS allows document authors to specify the presentation of elements on a Web page (spacing, margins, etc.) separately from the structure of the document (section headers, body text, links, etc.). Readers not familiar with these technologies will want to read Appendix I and Appendix J, which describe XHTML in detail and Appendix K, Cascading Style Sheets, which introduces CSS.

Figure 6.3 illustrates the full program listing for our first CGI script. Line 1

```
#!c:\Python\python.exe
```

is a *directive* (sometimes called the *pound-bang* or *sh-bang*) that specifies the location of the Python interpreter on the server. This directive must be the first line in a CGI script. The examples in this chapter are for Window users. For UNIX or Linux-based machines, the directive typically is one of the following:

```
#!/usr/bin/python
#!/usr/local/bin/python
#!/usr/bin/env python
```

depending on the location of the Python interpreter. [*Note*: If you do not know where the Python interpreter resides, contact the server administrator.]

Common Programming Error 6.1

*Forgetting to put the directive (**#!**) in the first line of a CGI script is an error if the Web server running the script does not understand the **.py** filename extension.*

Line 5 **import**s module ***time***. This module obtains the current time on the Web server and displays it in the user's browser. Lines 7–17 define function **printHeader**. This function takes argument **title**, which corresponds to the title of the Web page. Line

```
1  #!c:\Python\python.exe
2  # Fig. 6.3: fig06_03.py
3  # Displays current date and time in Web browser.
4
5  import time
6
7  def printHeader( title ):
8     print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12    "-//W3C//DTD XHTML 1.0 Strict//EN"
13    "DTD/xhtml1-strict.dtd">
14 <html xmlns = "http://www.w3.org/1999/xhtml">
15 <head><title>%s</title></head>
16
17 <body>""" % title
18
19 printHeader( "Current date and time" )
20 print time.ctime( time.time() )
21 print "</body></html>"
```

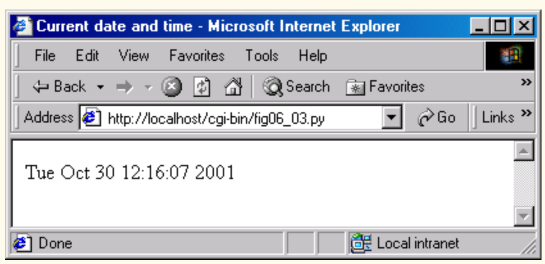


Fig. 6.3 CGI script displaying the date and time.

8 prints the HTTP header. Notice that line 9 is blank, which denotes the end of the HTTP headers. The line that follows the last HTTP header must be a blank line, otherwise Web browsers cannot render the content properly. Lines 10–14 print the XML declaration, document type declaration and opening **`<html>`** tag. For more information on XML, see Chapter 15. Lines 15–17 contain the XHTML document header and title and begin the XHTML document body.

Common Programming Error 6.2

Failure to place a blank line after an HTTP header is an error.

Line 19 begins the main portion of the program by calling function **`printHeader`** and passing an argument that represents the title of the Web page. Line 20 calls two functions in module **`time`** to print the current time. Function ***`time.time`*** returns a floating-point value that represents the number of seconds since midnight, January 1, 1970 (called

the *epoch*). Function ***time.ctime*** takes as an argument the number of seconds since the epoch and returns a human-readable string that represents the current time. We conclude the program by printing the XHTML body and document closing tags. For a complete list of functions in module **time**, visit

```
www.python.org/doc/current/lib/module-time.html
```

Note that the program consists almost entirely of **print** statements. Until now, the output of **print** has always displayed on the screen. However, technically speaking, the default target for **print** is *standard output*—an information stream presented to the user by an application. Typically, standard output is displayed on the screen, but it may be sent to a printer, written to a file, etc. When a Python program executes as a CGI script, the server redirects the standard output to the client Web browser. The browser interprets the headers and tags as if they were part of a normal server response to an XHTML document request.

Executing the program requires a properly configured server. [*Note*: In this book, we use the Apache Web server. For information on obtaining and configuring Apache, refer to our Python Web resources at **www.deitel.com**.] Once a server is available, the Web server site administrator specifies where CGI scripts can reside and what names are allowed for them. In our example, we place the Python file in the Web server's **cgi-bin** directory. For UNIX and Linux users, it also is necessary to set the permissions before executing the program. For example, UNIX and Linux command

```
chmod 755 fig06_02.py
```

gives the client the permission to read and execute **fig06_02.py**.

Assuming that the server is on the local computer, execute the program by typing

```
http://localhost/cgi-bin/fig06_02.py
```

in the browser's **Address** or **Location** field. If the server resides on a different computer, replace **localhost** with the server's hostname or IP address. [*Note*: The IP address of **localhost** is always **127.0.0.1**.] Requesting the document causes the server to execute the program and return the results.

Figure 6.4 illustrates the process of calling a CGI script. First, the client requests the resource named **fig06_02.py** from the server, just as the client requested **downloads.html** in the previous example (Step 1). If the server has not been configured to handle CGI scripts, it might return the Python code as text to the client.

A properly configured Web server, however, recognizes that certain resources need to be processed differently. For example, when the resource is a CGI script, the script must be executed by the Web server. A resource usually is designated as a CGI script in one of two ways—either it has a special filename extension (such as **.cgi** or **.py**), or it is located in a specific directory (often **cgi-bin**). In addition, the server administrator must grant explicit permission for remote access and CGI-script execution.

The server recognizes that the resource is a Python script and invokes Python to execute the script (Step 2). The program executes, and the text sent to standard output is returned to the Web server (Step 3). Finally, the Web server prints an additional line to the output that indicates the status of the HTTP transaction (such as **HTTP/1.1 200 OK**, for success) and sends the whole body of text to the client (Step 4).

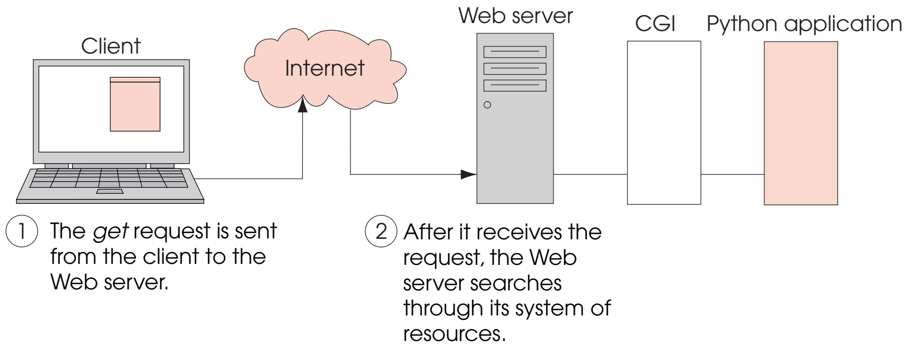


Fig. 6.4 Step 1: The **`GET`** request, **`GET /cgi-bin/fig06_02.py HTTP/1.1`**. (Part 1 of 4.)

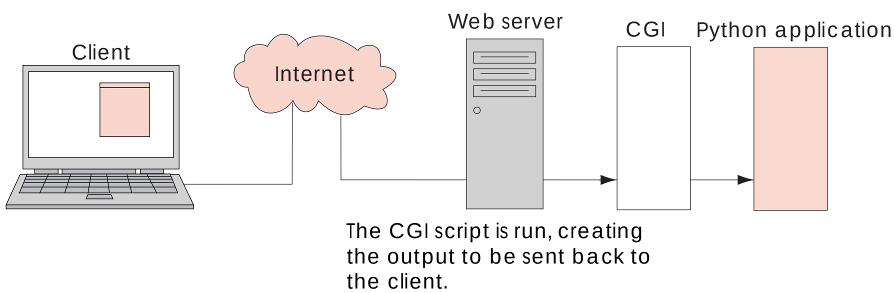


Fig. 6.4 Step 2: The Web server starts the CGI script. (Part 2 of 4.)

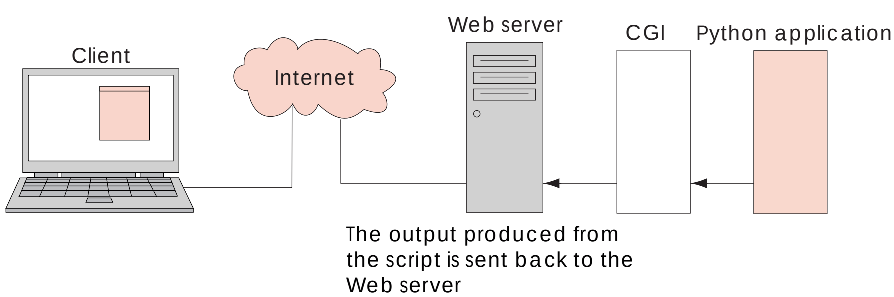


Fig. 6.4 Step 3: The output of the script is sent to the Web server. (Part 3 of 4.)

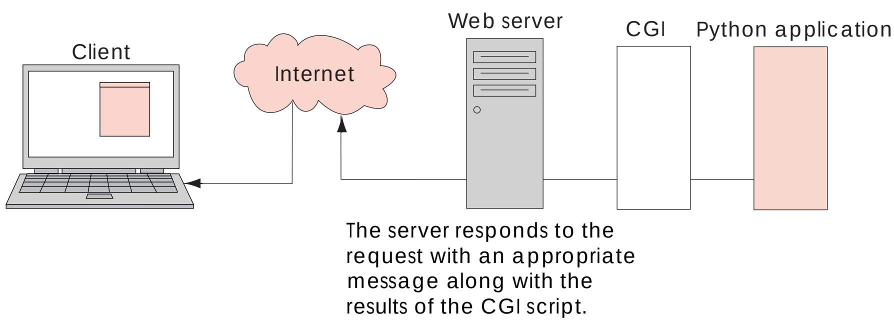


Fig. 6.4 Step 4: The HTTP response, **`HTTP/1.1 200 OK`**. (Part 4 of 4.)

The browser on the client side then processes the XHTML output and displays the results. It is important to note that the browser does not know about the work the server has done to execute the CGI script and return XHTML output. As far as the browser is concerned, it is requesting a resource like any other and receiving a response like any other. The client computer is not required to have a Python interpreter installed, because the script executes on the server. The client simply receives and processes the script's output.

We now consider a more involved CGI program. Figure 6.5 organizes all *CGI environment variables* and their corresponding values in an XHTML table, which is then displayed in a Web browser. Environment variables contain information about the execution environment in which script is being run. Such information includes the current user name and the name of the operating system. A CGI program uses environment variables to obtain information about the client (e.g., the client's IP address, operating system type, browser type, etc.) or to obtain information passed from the client to the CGI program.

Line 6 **`import`**s module ***`cgi`***. This module provides several CGI-related capabilities, including text-formatting, form-processing and URL parsing. In this example, we use module **`cgi`** to format XHTML text; in later examples, we use module **`cgi`** to process XHTML forms.

```
1  #!c:\Python\python.exe
2  # Fig. 6.5: fig06_05.py
3  # Program displaying CGI environment variables.
4
5  import os
6  import cgi
7
8  def printHeader( title ):
9     print """Content-type: text/html
10
11 <?xml version = "1.0" encoding = "UTF-8"?>
12 <!DOCTYPE html PUBLIC
13    "-//W3C//DTD XHTML 1.0 Strict//EN"
14    "DTD/xhtml1-strict.dtd">
15 <html xmlns = "http://www.w3.org/1999/xhtml">
16 <head><title>%s</title></head>
```

Fig. 6.5 CGI program to display environment variables. (Part 1 of 2.)

```
17
18 <body>""" % title
19
20 rowNumber = 0
21 backgroundColor = "white"
22
23 printHeader( "Environment Variables" )
24 print """<table style = "border: 0">"""
25
26 # print table of cgi variables and values
27 for item in os.environ.keys():
28    rowNumber += 1
29
30    if rowNumber % 2 == 0:           # even row numbers are white
31       backgroundColor = "white"
32    else:                            # odd row numbers are grey
33       backgroundColor = "lightgrey"
34
35    print """<tr style = "background-color: %s">
36    <td>%s</td><td>%s</td></tr>""" % ( backgroundColor,
37       cgi.escape( item ), cgi.escape( os.environ[ item ] ) )
38
39 print """</table></body></html>"""
```

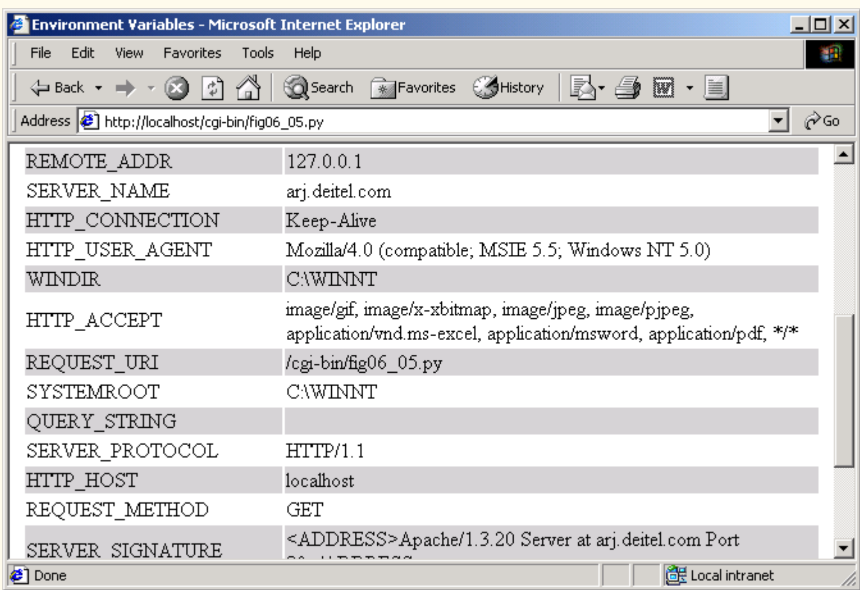


Fig. 6.5 CGI program to display environment variables. (Part 2 of 2.)

Lines 8–18 define function **printHeader**, which is identical to the function we defined in the previous example. The main program prints an XHTML table that contains the environment variables (lines 24–39). The ***os.environ*** data member holds all the environment variables (line 27). This data member acts like a dictionary; therefore, we can access its keys via the **keys** method and its values via the **[]** operator. Lines 30–33 set the

background color for each row. For each environment variable, lines 35–37 create a new row in the table containing that key and the corresponding value.

Note that line 37 calls function ***cgi.escape*** and passes as values each environment variable name and value. This function takes a string and returns a properly formatted XHTML string. Proper formatting means that special XHTML characters, such as the less-than and greater-than signs (**<** and **>**), are "escaped." For example, function **escape** returns a string where "**<**" is replaced by "**<**", "**>**" is replaced by "**>**" and "**&**" is replaced by "**&**". The replacement signifies that the browser should display a character instead of treating the character as markup. After we have printed all the environment variables, we close the **table**, **body** and **html** tags (line 39).

6.4 Sending Input to a CGI Script

You have seen one example of a CGI script processing preset environment variables. We now use an environment variable to supply data (e.g., client's name, search-engine query, etc.) to a CGI script. This section presents the environment variable ***QUERY_STRING*** that provides such a mechanism. The **QUERY_STRING** variable contains extra information that is appended to a URL in a **GET** request, following a question mark (**?**). For example, the URL

www._somesite_**.com/cgi-bin/script.py?state=California**

causes the Web browser to request a resource from **www.**_somesite_**.com**. The resource uses a CGI script (**cgi-bin/script.py**) to execute. The information following the **?** (**state=California**) is assigned by the Web server to the **QUERY_STRING** environment variable. Note that the question mark is not part of the resource requested, nor is it part of the query string; it serves as a *delimiter* (or separator) between the resource and the query string.

Figure 6.6 shows a simple example of a CGI script that reads and responds to data passed through the **QUERY_STRING** environment variable. The CGI script reading the string needs to know how to interpret the formatted data. In the example, the query string contains a series of name-value pairs separated by ampersands (**&**), as in

country=USA&state=California&city=Sacramento

Each name-value pair consists of a name (e.g., **country**) and a value (e.g., **USA**), delimited by an equal sign.

In line 24 of Fig. 6.6, we assign the value of environment-variable **QUERY_STRING** to variable **query**. Line 26 then tests to determine whether **query** is empty. If so, a message prints instructing the user to add a query string to the URL. We also provide a link to a URL that includes a sample query string. Note that query-string data may also be specified as part of a hypertext link in a Web page.

```
1  #!c:\Python\python.exe
2  # Fig. 6.6: fig06_06.py
3  # Example using QUERY_STRING.
4
5  import os
6  import cgi
```

Fig. 6.6 Reading input from **QUERY_STRING**. (Part 1 of 3.)

```

def printHeader( title ):
   print """Content-type: text/html

<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml">
<head><title>%s</title></head>

<body>""" % title

printHeader( "QUERY_STRING example" )
print "<h1>Name/Value Pairs</h1>"

query = os.environ[ "QUERY_STRING" ]

if len( query ) == 0:
   print """<p><br />
      Please add some name-value pairs to the URL above.
      Or try
      <a href = "fig06_06.py?name=Veronica&amp;age=23">this</a>.
      </p>"""
else:
   print """<p style = "font-style: italic">
      The query string is '%s'.</p>""" % cgi.escape( query )
   pairs = cgi.parse_qs( query )

   for key, value in pairs.items():
      print "<p>You set '%s' to value %s</p>"" % \
         ( key, value )

print "</body></html>"
```

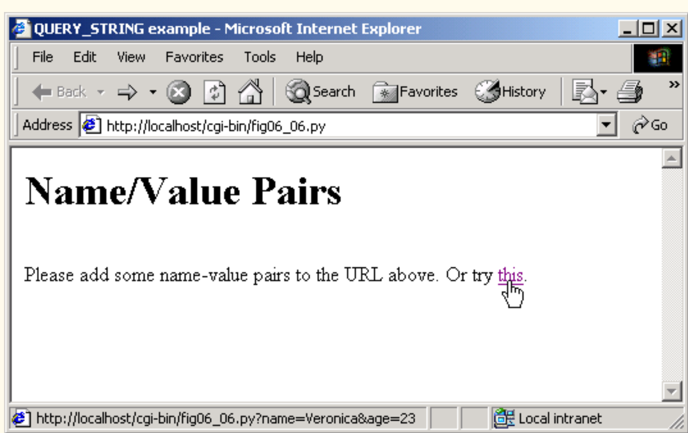


Fig. 6.6 Reading input from `QUERY_STRING`. (Part 2 of 3.)

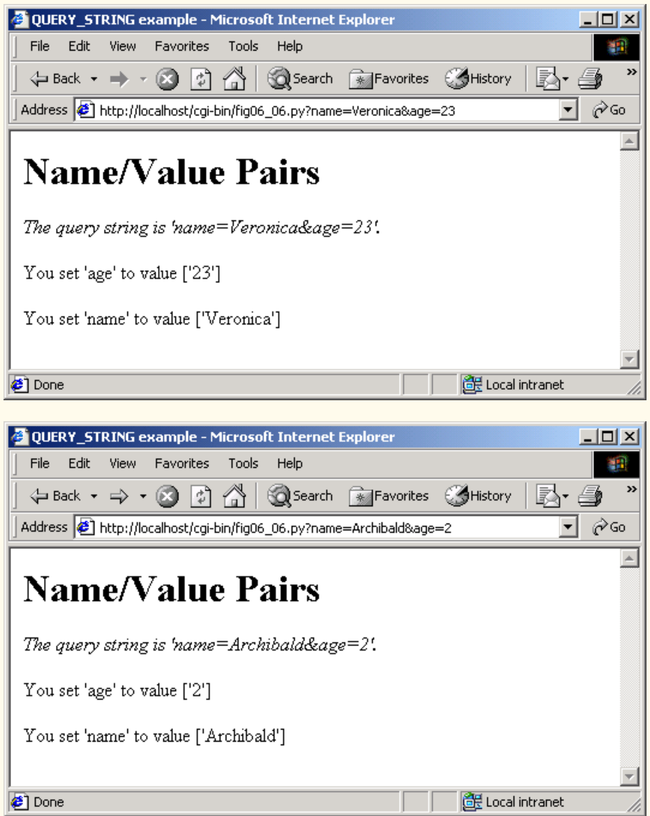


Fig. 6.6 Reading input from **`QUERY_STRING`**. (Part 3 of 3.)

If the query string is not empty, the value of the query string (lines 31–32) prints. Function ***`cgi.parse_qs`*** parses (i.e., "splits-up") the query string (line 33). This function takes as an argument a query string and returns a dictionary of name-value pairs contained in the query string. Lines 35–37 contain a **`for`** loop to print the names and values contained in dictionary **`pairs`**.

6.5 Using XHTML Forms to Send Input and Using Module `cgi` to Retrieve Form Data

If Web page users had to type all the information that the page required into the page's URL every time the user wanted to access the page, Web surfing would be quite a laborious task. XHTML provides *forms* on Web pages that provide a more intuitive way for users to input information to CGI scripts.

The ***`<form>`*** and ***`</form>`*** *tags* surround an XHTML form. The **`<form>`** tag typically takes two attributes. The first attribute is ***`action`***, which specifies the operation to perform when the user submits the form. For our purposes, the operation usually will be to call a CGI script to process the form data. The second attribute is **`method`**, which is either *get* or *post*. In this section, we show examples using both methods. An XHTML form may

contain any number of elements. Figure 6.7 gives a brief description of several possible elements to include.

Figure 6.8 demonstrates a basic XHTML form that uses the HTTP *get* method. Lines 21–26 output the form. Notice that the **method** attribute is *get* and the **action** attribute is **fig06_08.py** (i.e., the script calls itself to handle the form data once they are submitted—this is called a *postback*).

The form contains two input fields. The first is a single-line text field (**type = "text"**) with the name **word** (line 23). The second displays a button, labeled **Submit word**, to submit the form data (line 24).

The first time the script executes, **QUERY_STRING** should contain no value (unless the user has specifically appended a query string to the URL). However, once the user enters a word into the **word** text field and clicks the **Submit word** button, the script is called again. This time, the **QUERY_STRING** environment variable contains the name of the text-input field (**word**) and the user-entered value. For example, if the user enters the word **python** and clicks the **Submit word** button, **QUERY_STRING** would contain the value **"word=python"**.

Tag name	**type** attribute (for **<input>** tags)	Description
<input>	**button**	A standard push button.
	checkbox	Displays a checkbox that can be checked (true) or unchecked (false).
	file	Displays a text field and button so the user can specify a file to upload to a Web server. The button displays a file dialog that allows the user to select a file.
	hidden	Hides data information from clients so that hidden form data can be used only by the form handler on the server.
	image	The same as **submit**, but displays an image rather than a button.
	password	Like **text**, but each character typed appears as an asterisk (*****) to hide the input (for security).
	radio	Radio buttons are similar to checkboxes, except that only one radio button in a group of radio buttons can be selected at a time.
	reset	A button that resets form fields to their default values.
	submit	A push button that submits form data according to the form's **action**.
	text	Provides single-line text field for text input. This attribute is the default input type.
<select>		Drop-down menu or selection box. When used with the **<option>** tag, **<select>** specifies items to select.
<textarea>		Multiline area in which text can be input or displayed.

Fig. 6.7 XHTML form elements.

```
#!c:\Python\python.exe
# Fig. 6.8: fig06_08.py
# Demonstrates get method with an XHTML form.

import cgi

def printHeader( title ):
   print """Content-type: text/html

<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml">
<head><title>%s</title></head>

<body>""" % title

printHeader( "Using 'get' with forms" )
print """<p>Enter one of your favorite words here:<br /></p>
   <form method = "get" action = "fig06_08.py">
      <p>
      <input type = "text" name = "word" />
      <input type = "submit" value = "Submit word" />
      </p>
   </form>"""

pairs = cgi.parse()

if pairs.has_key( "word" ):
   print """<p>Your word is:
      <span style = "font-weight: bold">%s</span></p>""" \
      % cgi.escape( pairs[ "word" ][ 0 ] )

print "</body></html>"
```



Fig. 6.8 *get* used with an XHTML form. (Part 1 of 2.)



Fig. 6.8 *get* used with an XHTML form. (Part 2 of 2.)

Line 28 uses function ***cgi.parse*** to parse the form data. This function is similar to function **cgi.parse_qs**, except that **cgi.parse** parses the data from standard input (as opposed to the query string) and returns the name-value pairs in a dictionary.

Thus, during the second execution of the script, when the query string is parsed, line 28 assigns the returned dictionary to variable **pairs**. If dictionary **pairs** contains the key **"word"**, the user has submitted at least one word and the program prints the word(s) to the browser. The words are passed to function **cgi.escape** in case the input includes some special characters (such as **<**, **>** or a space). Lines 31–33 use CSS to display the result. CSS is discussed in Appendix K, Cascading Style Sheets (CSS). In Fig. 6.8, we see that the spaces in the address bar are replace by plus signs because Web browsers URL-encode XHTML-form data they send, which means that spaces are turned into plus signs and that certain other symbols (such as the apostrophe) are translated into their ASCII value in hexadecimal and preceded with a percent sign.

Using *get* with an XHTML form passes data to the CGI script in the same way that we saw in Fig. 6.6—through environment variables. Another way that CGI scripts interact with servers is via standard input and the *post* method. For comparison purposes, let us now reimplement the application of Fig. 6.8 using *post*. Notice that the code in the two figures is virtually identical. The XHTML form indicates that we are now using the *post* method to submit the form data (line 21).

```
1  #!c:\Python\python.exe
2  # Fig. 6.9: fig06_09.py
3  # Demonstrates post method with an XHTML form.
4
5  import cgi
6
7  def printHeader( title ):
8     print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12    "-//W3C//DTD XHTML 1.0 Strict//EN"
13    "DTD/xhtml1-strict.dtd">
```

Fig. 6.9 *post* used with an XHTML form. (Part 1 of 2.)

```
<html xmlns = "http://www.w3.org/1999/xhtml">
<head><title>%s</title></head>

<body>""" % title

printHeader( "Using 'post' with forms" )
print """<p>Enter one of your favorite words here:<br /></p>
   <form method = "post" action = "fig06_09.py">
      <p>
      <input type = "text" name = "word" />
      <input type = "submit" value = "Submit word" />
      </p>
   </form>"""

pairs = cgi.parse()

if pairs.has_key( "word" ):
   print """<p>Your word is:
      <span style = "font-weight: bold">%s</span></p>""" \
      % cgi.escape( pairs[ "word" ][ 0 ] )

print "</body></html>"
```

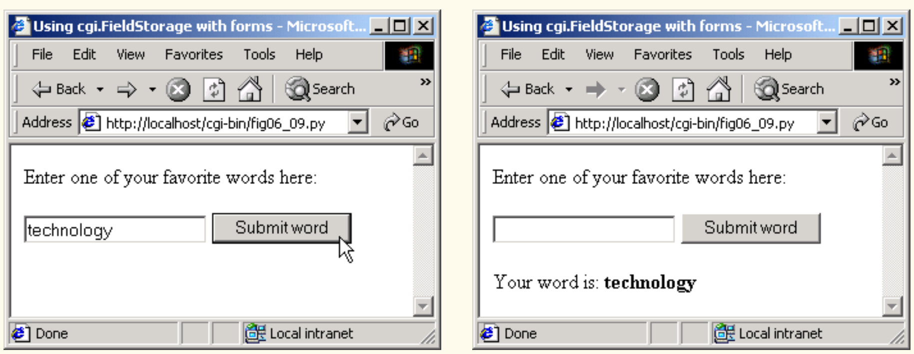


Fig. 6.9 *post* used with an XHTML form. (Part 2 of 2.)

The *post* method sends data to a CGI script via standard input. The data are encoded just as in **QUERY_STRING** (that is, with name-value pairs connected by equals signs and ampersands), but the **QUERY_STRING** environment variable is not set. Instead, the *post* method sets the environment variable ***CONTENT_LENGTH***, to indicate the number of characters of data that were sent (or posted). A benefit of the *post* method is that the number of characters of data can vary in size.

Although methods *get* and *post* are similar, some important differences exist. A *get* request sends form content as part of the URL. A *post* request posts form content to the end of an HTTP request. Another difference is the manner in which browsers process responses. Browsers often *cache* (save on disk) Web pages, so that when the Web page is requested a second time, the browser need not download the page again, but can load the page from the cache. This process speeds up the user's browsing experience by reducing the amount of data downloaded to view a Web page. Browsers do not cache the server

responses to *post* requests, however, because subsequent *post* requests might not contain the same information.

This method of handling responses is different from that of handling *get* requests. When a Web-based search engine is used, a *get* request normally supplies the search engine with the information specified in the XHTML form. The search engine then performs the search and returns the results as a Web page.

Software Engineering Observation 6.2

Most Web servers limit get request query strings to 1024 characters. If a query string exceeds this limit, use the post request.

Software Engineering Observation 6.3

Forms that contain many fields are submitted most often using a post *request. Sensitive form fields, such as passwords, usually are sent using* post *request.*

6.6 Using `cgi.FieldStorage` to Read Input

Figure 6.10 reimplements the example from Fig. 6.9 to take advantage of a high-level data abstraction provided by module **`cgi`**. Line 28 creates an object of class **`cgi.FieldStorage`**. [*Note*: Classes are discussed in Chapter 7, Object-Based Programming.] In our example, the high-level data type (or class) is called **`cgi.FieldStorage`** and resembles the dictionary returned by the parsing function.

```
1  #!c:\Python\python.exe
2  # Fig. 6.10: fig06_10.py
3  # Demonstrates use of cgi.FieldStorage an with XHTML form.
4
5  import cgi
6
7  def printHeader( title ):
8     print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12    "-//W3C//DTD XHTML 1.0 Strict//EN"
13    "DTD/xhtml1-strict.dtd">
14 <html xmlns = "http://www.w3.org/1999/xhtml">
15 <head><title>%s</title></head>
16
17 <body>""" % title
18
19 printHeader( "Using cgi.FieldStorage with forms" )
20 print """<p>Enter one of your favorite words here:<br /></p>
21    <form method = "post" action = "fig06_10.py">
22       <p>
23       <input type = "text" name = "word" />
24       <input type = "submit" value = "Submit word" />
25       </p>
26    </form>"""
27
```

Fig. 6.10 `cgi.FieldStorage` used with an XHTML form. (Part 1 of 2.)

```
28  form = cgi.FieldStorage()
29
30  if form.has_key( "word" ):
31     print """<p>Your word is:
32        <span style = "font-weight: bold">%s</span></p>""" \
33        % cgi.escape( form[ "word" ].value )
34
35  print "</body></html>"
```

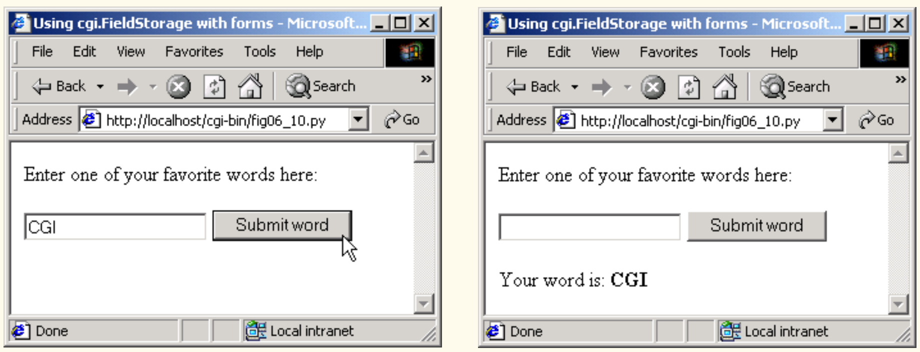


Fig. 6.10 `cgi.FieldStorage` used with an XHTML form. (Part 2 of 2.)

Line 30 calls dictionary method **`has_key`** and passes **`form`**, to determine whether the dictionary contains the key **`"word"`**. If so, the user has entered a word, and the program prints the word to the browser (lines 31–33). Note that, to access the value of any key in a **`cgi.FieldStorage`** object, we must access the value attribute of the key's corresponding value.

6.7 Other HTTP Headers

We mentioned at the close of Section 6.2.3 that there are alternatives to the standard HTTP header

```
Content-type: text/html
```

For example,

```
print "Content-type: text/plain"
```

prints the **`Content-type`** header with the **`text/plain`** content type. If the **`content-type`** of a page is specified as **`text/plain`**, the page is processed as plain text instead of as an HTML or XHTML document.

In addition to HTTP header **`Content-type`**, a CGI script can supply other HTTP headers. In most cases, the server passes these extra headers to the client untouched. For example, the following ***`Refresh`*** header redirects the client to a new location after a specified amount of time:

```
Refresh: "5; URL = http://www.deitel.com/newpage.html"
```

Five seconds after the Web browser receives this header, the browser requests the resource at the specified URL. Alternatively, the **Refresh** header can omit the URL, in which case it refreshes the current page at the given time interval.

The CGI protocol indicates that certain types of headers output by a CGI script are to be handled by the server, rather than be passed directly to the client. The first of these is the ***Location*** header. Like the **Refresh** header, **Location** redirects the client to a new location:

```
Location: http://www.deitel.com/newpage.html
```

If used with a *relative URL* (e.g., **Location: /newpage.html**), the **Location** header indicates to the server that the redirection is to be performed on the server side, without sending the **Location** header back to the client. In this case, it appears to the user as if the browser originally requested that resource. When a Python script uses the **Location** header, the **Content-type** header is not necessary because the new resource has its own content type.

The CGI specification also includes a ***Status*** header, which tells the server to output a status-header line (e.g., **HTTP/1.1 200 OK**). Normally, the server sends the appropriate status line to the client (adding, for example, the **200 OK** status code in most cases). However, CGI allows you to change the response status if you so desire. For example, sending a

```
Status: 204 No Response
```

header indicates that, although the request was successful, the browser should continue to display the same page. This header might be useful if you want to allow users to submit forms without moving to a new page.

We now have covered the fundamentals of the CGI protocol. To review, the CGI protocol allows scripts to interact with servers in three basic ways:

1. through the output of headers and content to the client via standard output;
2. by the server's setting of environment variables (including the URL-encoded **QUERY_STRING**) whose values are available within the script (via **os.environ**); and
3. through posted, URL-encoded data that the server sends to the script's standard input.

6.8 Example: Interactive Portal

Figure 6.11 and Fig. 6.12 show the implementation of a simple interactive portal (login page) for the fictional Bug2Bug Travel Web site. The example queries the client for a name and password, then displays information based on data entered. For simplicity, the example does not encrypt the data sent to the server.

Figure 6.11 displays the opening page. It is a static XHTML document containing a form that posts data to the **fig06_12.py** CGI script (line 14). The form contains one field to collect the client's name (line 17) and one to collect the member password (line 20). To make this XHTML file available from an Apache server, place **fig06_11.html** in the root directory of the Apache server (e.g., **Apache Group\Apache\htdocs**). For more information on Apache servers, visit **www.apache.org**.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">
<!-- Fig. 6.11: fig06_11.html    -->
<!-- Bug2Bug Travel log-in page. -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Enter here</title></head>

   <body>
      <h1>Welcome to Bug2Bug Travel</h1>

      <form method = "post" action = "/cgi-bin/fig06_12.py">

         <p>Please enter your name:<br />
         <input type = "text" name = "name" /><br />

         Members, please enter the password:<br />
         <input type = "password" name = "password" /><br />
         </p>

         <p style = "font-size: em - 1; font-style: italic" >
         Note that password is not encrypted.<br /><br />
         <input type = "submit" />
         </p>

      </form>
   </body>
</html>
```



Fig. 6.11 Interactive portal to create a password-protected Web page.

Figure 6.12 is the CGI script that processes the data received from the client. Line 20 retrieves the form data in a `cgi.FieldStorage` instance and assigns the result to local

variable **form**. The **if** structure that begins in line 22 tests whether **form** contains the key **"name"**. If **form** does not contain that key, the user has not entered a name, and we **print** a **Location** HTTP header (line 23) to redirect the user to the XHTML file where the user can enter a name (**fig06_11.html**). The document **fig06_11.html** is contained in the Web server's main document root (as indicated by the **/** that precedes the page name). The effect of line 23 is that clients who try to access **fig06_12.py** directly, without going through the login procedure, must enter through the portal.

```
1  #!c:\Python\python.exe
2  # Fig. 6.12: fig06_12.py
3  # Handles entry to Bug2Bug Travel.
4
5  import cgi
6
7  def printHeader( title ):
8     print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12    "-//W3C//DTD XHTML 1.0 Strict//EN"
13    "DTD/xhtml1-strict.dtd">
14 <html xmlns = "http://www.w3.org/1999/xhtml">
15 <head><title>%s</title></head>
16
17 <body>""" % title
18
19 form = cgi.FieldStorage()
20
21 if not form.has_key( "name" ):
22    print "Location: /fig06_11.html\n"
23 else:
24    printHeader( "Bug2Bug Travel" )
25    print "<h1>Welcome, %s!</h1>" % form[ "name" ].value
26    print """<p>Here are our weekly specials:<br /></p>
27       <ul><li>Boston to Taiwan for $300</li></ul>"""
28
29    if not form.has_key( "password" ):
30       print """<p style = "font-style: italic">
31          Become a member today for more great deals!</p>"""
32    elif form[ "password" ].value == "Coast2Coast":
33       print """<hr />
34          <p>Current specials just for members:<br /></p>
35          <ul><li>San Diego to Hong Kong for $250</li></ul>"""
36    else:
37       print """<p style = "font-style: italic">
38          Sorry, you have entered the wrong password.
39          If you have the correct password, enter
40          it to see more specials.</p>"""
41
42    print "<hr /></body></html>"
```

Fig. 6.12 Interactive portal handler. (Part 1 of 2.)



Fig. 6.12 Interactive portal handler. (Part 2 of 2.)

If a user has entered a name, we print a greeting that includes the user's name and the weekly specials (lines 26–28). Line 30 tests whether the user entered a password. If the user has not entered a password, we invite the user to become a member (line 31). If the user has entered a password, line 32 determines whether the password is equal to the string **`"Coast2Coast"`**. If true, we print the member specials to the browser. Note that the password, weekly specials and member specials are hard-coded (i.e., their values are supplied in the code). If the user-entered password does not equal **`"Coast2Coast"`**, the application requests the user to enter a valid password (lines 36–38).

Performance Tip 6.1

In response to each CGI request, a Web server executes a CGI program to create the response to the client. This process often takes more time than returning a static document. When implementing a Web site, define content that does not change frequently as static content. This practice allows the Web server to respond to clients more quickly than if only CGI scripting were used.

6.9 Internet and World Wide Web Resources

`www.w3.org/CGI`
The World Wide Web Consortium page on CGI is concerned with security issues involving the Common Gateway Interface. This page provides links to CGI specifications, as indicated by the National Center for Supercomputing Applications (NCSA).

`www.nacs.uci.edu/indiv/ehood/MIME/MIME.html`
This document provides links to MIME RFCs (Request for Comments), MIME related RFCs and other MIME-related information.

`www.speakeasy.org/~cgires`
This is a collection of tutorials and scripts related to CGI.

`www.fastcgi.com`
This is the home page of fast CGI—an extension to CGI that for high performance Internet applications

`bel-epa.com/pyapache`
This site is the resource center for **`PyApache`**. **`PyApache`** is a module that embeds the Python interpreter into the Apache server.

`www.modpython.org`
This is the home page of **`mod_python`**. Module **`mod_python`** is another module that embeds the Python interpreter within the Apache server. This module lets scripts run much faster than traditional CGI scripts.

SUMMARY

- The Common Gateway Interface (CGI) describes a set of protocols through which applications (commonly called CGI programs or CGI scripts) can interact with Web servers and (indirectly) with clients.
- The content of dynamic Web pages does not require modification by programmers, however the content of static Web pages requires modification by programmers.
- The Common Gateway Interface is "common" in the sense that it is not specific to any particular operating system (such as Linux or Windows) or to any one programming language.
- HTTP describes a set of methods and headers that allow clients and servers to interact and exchange information in a uniform and predictable way.

- A Web page in its simplest form is nothing more than an XHTML (Extensible Hypertext Markup Language) document. An XHTML document is just a plain-text file containing markings (markup, or tags) that describe to a Web browser how to display and format the information in the document.
- Hypertext information creates links to different pages or to other portions of the same page.
- Any XHTML file available for viewing over the Internet has a URL (Universal Resource Locator) associated with it. The URL contains information that directs a browser to the resource that the user wishes to access.
- The hostname is the name of the computer where a resource (such as an XHTML document) resides. The hostname is translated into an IP address, which identifies the server on the Internet.
- To request a resource, the browser first sends an HTTP request message to the server. The server responds with a line indicating the HTTP version, followed by a numeric code and a phrase describing the status of the transaction.
- The server normally sends one or more HTTP headers, which provide additional information about the data being sent. The header or set of headers is followed by a blank line, which indicates that the server has finished sending HTTP headers.
- Once the server sends the contents of the requested resource, the connection is terminated. The client-side browser processes the XHTML it receives and displays the results.
- *get* is an HTTP method that indicates that the client wishes to obtain a resource.
- The function **`time.ctime`**, when called with **`time.time()`**, returns a string value such as **`Wed Jul 18 10:54:57 2001`**.
- Redirecting output means sending output to somewhere other than the standard output, which is normally the screen.
- Just as standard input refers to the standard method of input into a program (usually the keyboard), standard output refers to the standard method of output from a program (usually the screen).
- If a server is not configured to handle CGI scripts, the server may return the Python program as text to display in a Web browser.
- A properly configured Web server will recognize a CGI script and execute it. A resource is usually designated as a CGI script in one of two ways: Either it has a specific filename extension or it is located in a specific directory. The server administrator must explicitly give permission for remote clients to access and execute CGI scripts.
- When the server recognizes that the resource requested is a Python script, the server invokes Python to execute the script. The Python program executes and the Web server sends the output to the client as the response to the request.
- With a CGI script, we must explicitly include the **`Content-type`** header, whereas, with an XHTML document, the header would be added by the Web server.
- The CGI protocol for output to be sent to a Web browser consists of printing to standard output the **`Content-type`** header, a blank line and the data (XHTML, plain text, etc.) to be output.
- Module **`cgi`** provides functions that simplify the creation of CGI scripts. Among other things, **`cgi`** includes a set of functions to aid in dynamic XHTML generation.
- The **`os.environ`** dictionary contains the names and values of all the environment variables.
- CGI-enabled Web servers set environment variables that provide information about both the server's and the client's script-execution environment.
- The environment variable **`QUERY_STRING`** provides a mechanism that enables programmers to supply any type of data to CGI scripts. The **`QUERY_STRING`** variable contains extra information that is appended to a URL, following a question mark (**`?`**). The question mark is not part of the resource requested or of the query string. It simply serves as a delimiter.

- Data put into a query string can be structured in a variety of ways, provided that the CGI script that reads the string knows how to interpret the formatted data.
- Forms provide another way for users to input information that is sent to a CGI script.
- The **<form>** and **</form>** tags surround an XHTML form.
- The **<form>** tag generally takes two attributes. The first attribute is **action**, which specifies the action to take when the user submits the form. The second attribute is **method**, which is either *get* or *post*.
- Using *get* with an XHTML form causes data to be passed to the CGI script through environment variable **QUERY_STRING**, which is set by the server.
- Web browsers URL-encode XHTML-form data that they send. This means that spaces are turned into plus signs and that certain other symbols (such as the apostrophe) are translated into their ASCII value in hexadecimal and preceded with a percent sign.
- A CGI script can supply HTTP headers in addition to **Content-type**. In most cases, the server passes these extra headers to the client untouched.
- The **Location** header redirects the client to a new location. If used with a relative URL, the **Location** header indicates to the server that the redirection is to be performed without sending the **Location** header back to the client.
- The CGI specification also includes a **Status** header, which informs the server to output a corresponding status header line. Normally, the server adds the appropriate status line to the output sent to the client. However, CGI allows users to change the response status.

TERMINOLOGY

#! directive
? in query string
127.0.0.1 IP address
action attribute
button attribute
protocol
CSS (Cascading Style Sheet)
CGI (Common Gateway Interface)
CGI environment variable
.cgi file extension
cgi module
CGI Script
cgi module
cgi.escape function
cgi.FieldStorage object
cgi.parse function
cgi.parse_qs function
cgi-bin directory
checkbox attribute value (**type**)
CONTENT_LENGTH
Content-type HTTP header
domain name system (DNS)
dynamic Web content
environment variable
file attribute value (**type**)
form XHTML element (**<form>…</form>**)
get method
HTTP header
hidden attribute value (**type**)
HTML (Hypertext Markup Language)
HTTP (Hypertext Transfer Protocol)
HTTP method
HTTP transaction
image attribute value (**type**)
image/gif MIME type
input HTML element
IP (Internet Protocol) address
localhost
Location HTTP header
method of XHTML form
MIME (Multipurpose Internet Mail Extensions)
os.environ data member
password attribute value (**type**)
.py file extension
post method
portal
pound-bang directive
QUERY_STRING environment variable
radio attribute value (**type**)
redirect
Refresh HTTP header
relative URL

reset attribute value (**type**)
select XHTML element (**form**)
sh-bang directive (**#!**)
static Web content
Status HTTP header
submit attribute value (**type**)
text attribute value (**type**)
text/html MIME type
text/txt MIME type
textarea XHTML element
time module
time.ctime function
time.time function
title XHTML element (**<title>…</title>**)
URL (Universal Resource Locator)
value attribute of **cgi.FieldStorage** object
virtual URL
document root
XHTML (Extensible Hypertext Markup Language)
XHTML form
XHTML tag

SELF-REVIEW EXERCISES

6.1 Fill in the blanks in each of the following statements:

a) CGI is an acronym for __________.
b) HTTP describes a set of __________ and __________ that allow clients and servers to interact.
c) The translation of a hostname into an IP address normally is performed by a __________.
d) The __________, which is part of the HTTP header sent with every type of data, helps the browser determine how to process the data it receives.
e) __________ are reserved memory locations that an operating systems maintains to keep track of system information.
f) Function _________ takes a string and returns a properly formatted XHTML string.
g) Variable __________ contains extra information that is appended to a URL in a *get* request, following a question mark.
h) The default target for **print** is __________.
i) The __________ data member contains all the environment variables.
j) XHTML __________ allow users to input information to a CGI script.

6.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The CGI protocol is not specific to any particular operating system or programming language.
b) Function **time.ctime** returns a floating-point value that represents the number of seconds since the epoch.
c) The first directive of a CGI script provides the location of the Python interpreter.
d) The forward slash character acts as a delimiter between the resource and the query string in a URL.
e) CGI scripts are executed on the client's machine.
f) The **Status: 204 No Response** header indicates that a request to the server failed.
g) Redirection sends output to somewhere other than the screen.
h) The **action** attribute of the **form** element specifies the action to take when the user submits the form.
i) A *post* request posts form contents to the end of an HTTP request.
j) Form data can be stored in an object of class **cgi.FormStorage**.

ANSWERS TO SELF REVIEW EXERCISES

6.1 a) Common Gateway Interface. b) methods, headers. c) domain name server (DNS). d) MIME type. e) Environment variables. f) **cgi.escape**. g) **QUERY_STRING**. h) standard output. i) **os.environ**. j) forms.

6.2 a) True. b) False. Function **`ctime.time`** takes a floating-point value that represents the number of seconds since the epoch as an argument and returns a human-readable string representing the current time. c) True. d) False. A question mark acts as a delimiter between the resource and the query string in a URL. e) False. The server executes CGI scripts. f) False. The **`Status: 204 No Response`** header indicates that, although the request was successful, the browser should continue to display the same page. g) True. h) True. i) True. j) False. Form data can be stored in an object of class **`cgi.FieldStorage`**.

EXERCISES

6.3 Write a CGI script that prints the squares of the integers from 1 to 10 on separate lines.

6.4 Modify your solution to Exercise 6.3 to display its output in an XHTML table. The left column should be the number, and the right column should be the square of that number.

6.5 Write a CGI script that receives as input three numbers from the client and returns a statement indicating whether the three numbers could represent an equilateral triangle (all three sides are the same length), an isosceles triangle (two sides are the same length) or a right triangle (the square of one side is equal to the sum of the squares of the other two sides).

6.6 Write a soothsayer CGI program that allows the user to submit a question. When the question is submitted, the server should display a random response from a list of vague answers.

6.7 You are provided with a portal page (see the code and output below) where people can buy products. Write the CGI script to enable this interactive portal. The user should specify how many of each item to buy. The total cost of the items purchased should be displayed to the user.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- Exercise 6.7: ex06_07.html                               -->
<!-- Interactive portal that compiles shopping list based -->
<!-- on user input.                                           -->

<html>
   <head>
      <title>Buy Something</title>
   </head>

   <body>
      <h1>Clearance!</h1>
      <p>Please enter how many of each product you would like to
        order into the box in the right-hand column.</p>

      <form method = "post" action =
               "http://localhost/cgi-bin/ex06_07.py">

         <table width = "100%" border = "3">
            <tr>
               <th>Product Name</th>
               <th>Description</th>
               <th>Price</th>
               <th>Order</th>
            </tr>

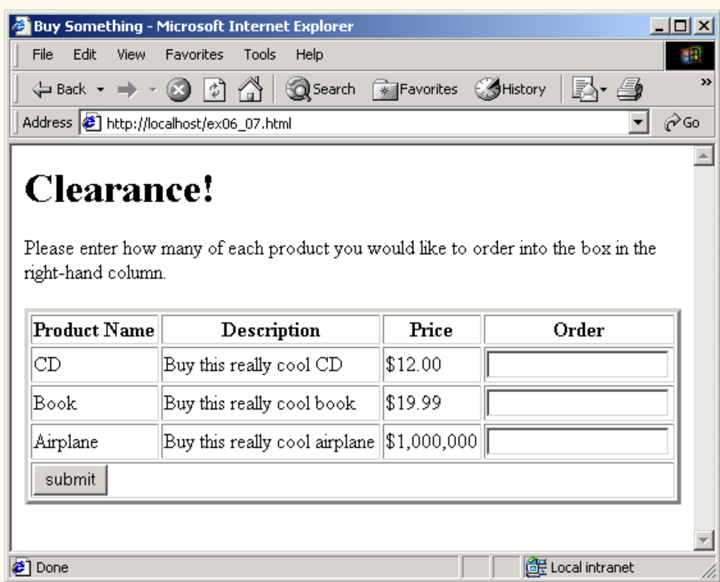
```

```
            <tr>
               <td>CD</td>
               <td>Buy this really cool CD</td>
               <td>$12.00</td>
               <td><input type = "text" name = "CD" /></td>
            </tr>

            <tr>
               <td>Book</td>
               <td>Buy this really cool book</td>
               <td>$19.99</td>
               <td><input type = "text" name = "book" /></td>
            </tr>

            <tr>
               <td>Airplane</td>
               <td>Buy this really cool airplane</td>
               <td>$1,000,000</td>
               <td><input type = "text" name = "airplane" /></td>
            </tr>
         </table>

        <input type = "submit" value = "submit">
     </form>
  </body>
</html>
```



6.8 Write a CGI script for a TV show survey. List five TV shows, let the survey participant rank the TV shows with numbers from 1 (least favorite) to 5 (most favorite). Display the participant's most favorite TV show.

7

Object-Based Programming

Objectives

- To understand the software-engineering concepts of "encapsulation" and "data hiding."
- To understand the notions of data abstraction and abstract data types (ADTs).
- To create Python ADTs, namely classes.
- To understand how to create, use and destroy objects of a class.
- To control access to object attributes and methods.
- To begin to appreciate the value of object orientation.

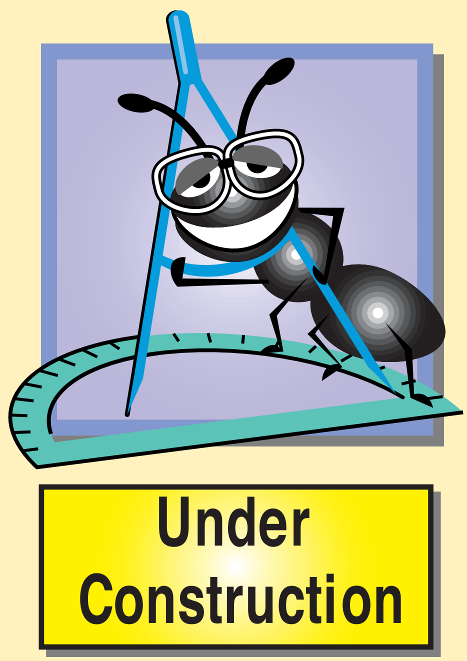
My object all sublime
I shall achieve in time.
W. S. Gilbert

Is it a world to hide virtues in?
William Shakespeare, *Twelfth Night*

Your public servants serve you right.
Adlai Stevenson

Classes struggle, some classes triumph, others are eliminated.
Mao Zedong

This above all: to thine own self be true.
William Shakespeare, *Hamlet*



Outline

7.1 Introduction

Now we begin our deeper study of object orientation. Through our discussion of Python programs in Chapters 2–6, we have already encountered many basic concepts (i.e., "object think") and terminology (i.e., "object speak"). Let us briefly overview some key concepts and terminology of object orientation. *Object-oriented programming (OOP) encapsulates* (i.e., wraps) data (*attributes*) and functions (*behaviors*) into components called *classes*. The data and functions of a class are intimately tied together. A class is like a blueprint. Using a blueprint, a builder can build a house. Using a class, a programmer can create an *object* (also called an *instance*). One blueprint can be reused many times to make many houses. One class can be reused many times to make many objects of the same class. Classes have a property called *information hiding*. This means that, although objects may know how to communicate with one another across well-defined *interfaces*, one object normally should not be allowed to know how another object is implemented—implementation details are hidden within the objects themselves. Surely it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is crucial to good software engineering.

In C and other *procedural programming languages*, programming tends to be *action-oriented*; in Python, programming can be *object-oriented*. In procedural programming, the unit of programming is the *function*. In object-oriented programming, the unit of programming is the *class* from which objects eventually are *instantiated* (i.e., created).

Procedural programmers concentrate on writing functions. Groups of actions that perform some task are formed into functions, and functions are grouped to form programs. Data certainly is important in procedural programming, but the view is that data exists primarily in support of the actions that functions perform. The *verbs* in a system specification—a document that describes the services an application should provide—help the

procedural programmer determine the set of functions that will work together to implement the system.

Object-oriented programmers concentrate on creating their own *user-defined types*, called *classes.* Classes are also referred to as *programmer-defined types*. Each class contains data and the set of functions that manipulate the data. The data components of a class are called *attributes* (or *data members*). The functional components of a class are called *methods* (or *member functions*, in other object-oriented languages). The focus of attention in object-oriented programming is on classes rather than on functions. The *nouns* in a system specification help the object-oriented programmer determine the set of classes that will be used to create the objects that will work together to implement the system.

Software Engineering Observation 7.1

A central theme of this book is "reuse, reuse, reuse." We will carefully discuss a number of techniques for "polishing" classes to encourage reuse. We focus on "crafting valuable classes" and creating valuable "software assets."

7.2 Implementing a Time Abstract Data Type with a Class

Classes enable programmers to model objects that have data (represented as attributes) and behaviors—or *operations*—(represented as methods). Methods are invoked in response to *messages* sent to objects. A message corresponds to a method call sent from one object to another.

Classes simplify programming because the *clients* (or users of the class) need to be concerned only with the operations encapsulated or embedded in the object—the object interface. Such operations usually are designed to be client-oriented rather than implementation-oriented. Clients do not need to be concerned with a class's implementation (although clients, of course, want correct and efficient implementations). When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation eliminates the possibility of other program parts becoming dependent on the details of the class implementation.

Often, classes do not have to be created "from scratch." Rather, they may be *derived* from other classes that provide attributes and behaviors the new classes can use—or classes can include objects of other classes as members. Such *software reuse* can greatly enhance programmer productivity. Deriving new classes from existing classes is called *inheritance* and is discussed in detail in Chapter 9, Object-Oriented Programming: Inheritance.

Figure 7.1 contains a simple definition for class **`Time`**. The class contains information that describes the time of day and contains methods for printing the time in two formats. The class maintains the time internally in a 24-hour format (i.e., military time), but allows the client to display the time in either 24-hour format or in "standard" (AM, PM) format. Later in this section, we present a program (Fig. 7.2) that demonstrates how to create an object of class **`Time`**.

Keyword ***`class`*** (line 4) begins a class definition. The keyword is followed by the name of the class (**`Time`**), which is followed by a colon (**`:`**). The line that contains keyword **`class`** and the class name is called the class's *header*. The *body* of the class is an indented code block (lines 5–37) that contains methods and attributes that belong to the class. Class names usually follow the same naming conventions as variable names, except that the first word of the class name is capitalized.

```
1  # Fig. 7.1: Time1.py
2  # Simple definition of class Time.
3
4  class Time:
5     """Time abstract data type (ADT) definition"""
6
7     def __init__( self ):
8        """Initializes hour, minute and second to zero"""
9
10       self.hour = 0      # 0-23
11       self.minute = 0    # 0-59
12       self.second = 0    # 0-59
13
14    def printMilitary( self ):
15       """Prints object of class Time in military format"""
16
17       print "%.2d:%.2d:%.2d" % \
18          ( self.hour, self.minute, self.second ),
19
20    def printStandard( self ):
21       """Prints object of class Time in standard format"""
22
23       standardTime = ""
24
25       if self.hour == 0 or self.hour == 12:
26          standardTime += "12:"
27       else:
28          standardTime += "%d:" % ( self.hour % 12 )
29
30       standardTime += "%.2d:%.2d" % ( self.minute, self.second )
31
32       if self.hour < 12:
33          standardTime += " AM"
34       else:
35          standardTime += " PM"
36
37       print standardTime,
```

Fig. 7.1 **Time** class—contains attributes and methods for storing and displaying time of day.

Common Programming Error 7.1

Failure to include a colon at the end of a class definition header is a syntax error.

Common Programming Error 7.2

Failure to indent the body of a class is a syntax error.

Line 5 contains the class's optional *documentation string*—a string that describes the class. If a class contains a documentation string, the string must appear in the line or lines following the class header. A user can view a class's documentation string by executing the following statement

```
print ClassName.__doc__
```

Modules, methods and functions also may specify a documentation string.

Good Programming Practice 7.1

Include documentation strings, where appropriate, to enhance program clarity.

Good Programming Practice 7.2

By convention, docstrings are triple-quoted strings. This convention allows the class author to expand a program's documentation (e.g., by adding several more lines) without having to change the quote style.

Line 7 begins the definition for special method `__init__`, the *constructor* method of the class. A constructor is a special method that executes each time an object of a class is created. The constructor (method `__init__`) initializes the attributes of the object and returns `None`. Python classes may define several other special methods, identified by leading and trailing double-underscores (`__`) in the name. We discuss many of these special methods in Chapter 8, Customizing Classes.

Common Programming Error 7.3

Returning a value other than `None` from a constructor is a fatal, runtime error.

Software Engineering Observation 7.2

Ensure that objects are initialized before client code invokes those objects' methods. Do not rely on client code to initialize objects properly.

Good Programming Practice 7.3

When appropriate (almost always), provide a constructor to ensure that every object is initialized with meaningful values.

All methods, including constructors, must specify at least one parameter. This parameter represents the object of the class for which the method is called. This parameter often is referred to as the *class instance object*. This term can be confusing, so we refer to the first argument of any method as the *object reference argument*, or simply the *object reference*. Methods must use the object reference to access attributes and other methods that belong to the class. By convention, the object reference argument is called `self`.

Common Programming Error 7.4

Failure to specify an object reference (usually called `self`) as the first parameter in a method definition causes fatal logic errors when the method is invoked at runtime.

Good Programming Practice 7.4

Name the first parameter of all methods `self`. This naming convention helps ensure conformity across Python programs written by different programmers.

Each object has its own namespace that contains the object's methods and attributes. The class's constructor starts with an empty object (`self`) and adds attributes to the object's namespace. For example, the constructor for class `Time` (lines 7–12) adds three attributes (`hour`, `minute` and `second`) to the new object's namespace. Line 10 binds attribute `hour` to the object's namespace and initializes the attribute's value to 0. Once an attribute has been added to an object's namespace, a client that uses the object may access the attribute's value.

Class **`Time`** also defines methods **`printMilitary`** and **`printStandard`**. Notice that methods can specify a docstring, in the line or lines following the method header. In this example, each method specifies one parameter (**`self`**) that refers to the object of the class for which the method is invoked. Each method accesses the object's attributes through parameter **`self`**. Method **`printMilitary`** (lines 14–18) prints the time in military (24-hour) format. Method **`printStandard`** (lines 20–37) prints the time in standard (12-hour) format.

Once a class has been defined, programs can create objects of that class. Many objects of a class can exist and programmers can create objects as necessary. This is one reason why Python is said to be an *extensible language*. The program in Fig. 7.2 creates an object of class **`Time`**, defined in Fig. 7.1. We first import the class definition from file **`Time1.py`**—the file that contains the class definition. Line 4 imports the definition in the same way a program would import any element from a module.

```
1  # Fig. 7.2: fig07_02.py
2  # Creating and manipulating objects of class Time.
3
4  from Time1 import Time   # import class definition from file
5
6  time1 = Time()  # create object of class Time
7
8  # access object's attributes
9  print "The attributes of time1 are: "
10 print "time1.hour:", time1.hour
11 print "time1.minute:", time1.minute
12 print "time1.second:", time1.second
13
14 # access object's methods
15 print "\nCalling method printMilitary:",
16 time1.printMilitary()
17
18 print "\nCalling method printStandard:",
19 time1.printStandard()
20
21 # change value of object's attributes
22 print "\n\nChanging time1's hour attribute..."
23 time1.hour = 25
24 print "Calling method printMilitary after alteration:",
25 time1.printMilitary()
```

```
The attributes of time1 are:
time1.hour: 0
time1.minute: 0
time1.second: 0

Calling method printMilitary: 00:00:00
Calling method printStandard: 12:00:00 AM

Changing time1's hour attribute...
Calling method printMilitary after alteration: 25:00:00
```

Fig. 7.2 Creating an object.

One of the fundamental principles of good software engineering is that a client should not need to know how a class is implemented to use that class. Python's use of modules facilitates this data abstraction—the program in Fig. 7.2 simply imports the **`Time`** definition and uses class **`Time`** without knowing how the class is implemented.

Software Engineering Observation 7.3

Clients of a class do nost need access to the class's source code to use the class.

To create an object of class **`Time`**, simply "call" the class name as if it were a function (line 6). This call invokes the constructor for class **`Time`**. Even though the class definition stipulates that the constructor (**`__init__`**) takes one argument, line 6 does not pass any arguments to the constructor. Python inserts the first (object reference) argument into every method call, including a class's constructor call. The constructor initializes the object's attributes. Once the constructor exits, Python assigns the newly created object to **`time1`**.

Client code must access an object's attributes through a reference to that object. Lines 10–12 demonstrate how a program can access an object's attributes through the dot (**`.`**) access operator. The name of the object appears to the left of the dot, and the attribute appears to the right of the dot. The output demonstrates the initial values that the constructor assigned to attributes **`hour`**, **`minute`** and **`second`**.

Client code can access an object's methods in a similar manner. Line 16 calls **`time1`**'s **`printMilitary`** method. Notice again that the method call passes no arguments, even though the method definition specifies one parameter called **`self`**. Python passes a reference to **`time1`** in the **`printMilitary`** call, so the method may access the object's attributes.

Line 23 modifies the value assigned to attribute **`time1.hour`**. The output from lines 24–25 shows a problem that often arises when a client indiscriminately accesses an object's data. The meaning of attribute **`hour`** is unclear, because that data member now has a value of 25. We say that the data member is in an *inconsistent state* (it contains an invalid value). Some other programming languages provide ways to prevent a client from accessing an object's data. Python, on the other hand, does not provide such strict programming constructs. Later in this chapter, we discuss the various ways Python programmers ensure that an object's data remains in a consistent state.

Common Programming Error 7.5

Directly accessing an object's attributes may cause the data to enter an inconsistent state.

7.3 Special Attributes

Classes and objects of classes both have special attributes that can be manipulated. These attributes, which Python creates when a class is defined or when an object of a class is created, provide information about the class or object of a class to which they belong. Figure 7.3 lists the special attributes that all classes contain. The interactive session in Fig. 7.4 prints the value of each of these attributes for class **`Time`**.

Additionally, all objects of classes have attributes in common. Figure 7.5 lists these attributes, and the interactive session in Fig. 7.6 prints the attributes' values for an object of class **`Time`**. Notice that objects can access the **`__doc__`** and **`__module__`** attributes that belong to the object's class.

Attribute	Description
`__bases__`	A tuple that contains base classes from which the class directly inherits. If the class does not inherit from other classes, the tuple is empty. [*Note:* We discuss base classes and inheritance in Chapter 9, Object-Oriented Programming: Inheritance.]
`__dict__`	A dictionary that corresponds to the class's namespace. Each key-value pair represents an identifier and its value in the namespace.
`__doc__`	A class's docstring. If the class does not specify a docstring, the value is **`None`**.
`__module__`	A string that contains the module (file) name in which the class is defined.
`__name__`	A string that contains the class's name.

Fig. 7.3 Special attributes of a class.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>>
>>> from Time1 import Time
>>> print Time.__bases__
()
>>> print Time.__dict__
{'printMilitary': <function printMilitary at 0x0079BF80>,
'__module__': 'Time1', '__doc__': 'Time abstract data type (ADT)
definition', '__init__': <function __init__ at 0x0077AB00>,
'printStandard': <function printStandard at 0x00769990>}
>>>
>>> print Time.__doc__
Time abstract data type (ADT) definition
>>> print Time.__module__
Time1
>>> print Time.__name__
Time
```

Fig. 7.4 Special attributes of a class.

Attribute	Description
`__class__`	A reference to the class from which the object was instantiated.
`__dict__`	A dictionary that corresponds to the object's namespace. Each key-value pair represents an identifier and its value in the namespace.

Fig. 7.5 Special attributes of an object of a class.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>>
>>> from Time1 import Time
>>> time1 = Time()
>>> print time1.__class__
Time1.Time
>>> print time1.__dict__
{'second': 0, 'minute': 0, 'hour': 0}
>>> print time1.__doc__
Time abstract data type (ADT) definition
>>> print time1.__module__
Time1
```

Fig. 7.6 Special attributes of an object.

These attributes contribute to Python's strong *introspection* capabilities (i.e., Python's ability to provide information about itself). Many programmers use these capabilities to create advanced, dynamic and flexible applications. In this text, we use these capabilities mostly to explore how Python works and to further our understanding of programming concepts.

7.4 Controlling Access to Attributes

In this chapter we already have discussed how clients can access an object's attributes directly and how this practice can place an object's data in an inconsistent state. Most object-oriented programming languages allow an object to prevent its clients from accessing the object's data directly. However, in Python, the programmer uses attribute naming conventions to hide data from clients. In this section, we discuss the advantages and disadvantages of this practice.

7.4.1 Get and Set Methods

Although a client can access an object's data directly (and perhaps cause the data to enter an inconsistent state), a programmer can design classes to encourage correct use. One technique is for the class to provide *access methods* through which the data of the class can be read and written in a carefully controlled manner.

Predicate methods are read-only access methods that test the validity of a condition. An example of a predicate method is an **isEmpty** method for a *container* class—a class capable of holding many objects. A program calls **isEmpty** before reading another item from the container object. An **isFull** predicate method tests a container object to determine whether it has additional space in which a program can place an item. Some appropriate predicate methods for our **Time** class might be **isAM** and **isPM**.

When a class defines access methods, a client should access an object's attributes only through those access methods. A typical manipulation might be the adjustment of a customer's bank-account balance (e.g., an attribute of an object of class **BankAccount**) by a method **computeInterest**.

Classes often provide methods that allow clients to *set* (write) or *get* (read) the values of attributes. Although these methods need not be called *set* and *get*, they often are, by conven-

tion. More specifically, a method that *set*s data member **`interestRate`** typically would be named **`setInterestRate`**, and a method that *get*s the **`interestRate`** typically would be named **`getInterestRate`**. *Get* methods also are called "query" methods.

It may seem that providing both *set* and *get* capabilities provides no benefit over accessing the attributes directly, but there is a subtle difference. A *get* method seems to allow clients to read the data at will, but the *get* method can control the formatting of the data. A *set* method can—and most likely should—scrutinize attempts to modify the value of the attribute. This ensures that the new value is appropriate for that data item. For example, a set method can reject the following values: the value 37 as the date, a negative value as a person's body weight and the value 185 on an exam (when the grade range is 0–100).

Software Engineering Observation 7.4

Controlling access, especially write access, to attributes through access methods helps ensure data integrity.

Testing and Debugging Tip 7.1

Data integrity is not automatic, even if the programmer provides access methods—the programmer must provide for validity checking.

A class's *set* methods sometimes return values that indicate attempts were made to assign invalid data to an object of the class. Clients of the class then test the return values of *set* methods to determine whether the object it is manipulating is a valid object and to take appropriate actions if the object is not valid. Alternatively, a *set* method may specify that an error message—called an *exception*—be sent ("raised") to the client when the client attempts to assign an invalid value to an attribute. *Raising exceptions* is a topic we explore in detail in Chapter 12, Exception Handling. Exceptions are the preferred technique for handling invalid attribute values in Python.

Good Programming Practice 7.5

Methods that set *the values of data should verify that the intended new values are proper. If they are not, the* set *methods should indicate that an error has occurred.*

Software Engineering Observation 7.5

Accessing data through set *and* get *methods not only protects the data from assuming invalid values, but also insulates clients of the class from the representation of the data. Thus, if the representation of the data changes (typically to reduce the amount of storage required or to improve performance), only the method bodies need to change—the clients need not change as long as the interface provided by the methods remains the same.*

Figure 7.7—**`Time2.py`**—defines a modified **`Time`** class that uses access methods to protect access to the data stored in the class.

```
# Fig: 7.7: Time2.py
# Class Time with accessor methods.

class Time:
   """Class Time with accessor methods"""

```

Fig. 7.7 Access methods defined for class **`Time`**. (Part 1 of 3.)

```python
   def __init__( self ):
      """Time constructor initializes each data member to zero"""

      self._hour = 0     # 0-23
      self._minute = 0   # 0-59
      self._second = 0   # 0-59

   def setTime( self, hour, minute, second ):
      """Set values of hour, minute, and second"""

      self.setHour( hour )
      self.setMinute( minute )
      self.setSecond( second )

   def setHour( self, hour ):
      """Set hour value"""

      if 0 <= hour < 24:
         self._hour = hour
      else:
         raise ValueError, "Invalid hour value: %d" % hour

   def setMinute( self, minute ):
      """Set minute value"""

      if 0 <= minute < 60:
         self._minute = minute
      else:
         raise ValueError, "Invalid minute value: %d" % minute

   def setSecond( self, second ):
      """Set second value"""

      if 0 <= second < 60:
         self._second = second
      else:
         raise ValueError, "Invalid second value: %d" % second

   def getHour( self ):
      """Get hour value"""

      return self._hour

   def getMinute( self ):
      """Get minute value"""

      return self._minute

   def getSecond( self ):
      """Get second value"""

      return self._second

```

Fig. 7.7 Access methods defined for class `Time`. (Part 2 of 3.)

```
60      def printMilitary( self ):
61         """Prints Time object in military format"""
62
63         print "%.2d:%.2d:%.2d" % \
64            ( self._hour, self._minute, self._second ),
65
66      def printStandard( self ):
67         """Prints Time object in standard format"""
68
69         standardTime = ""
70
71         if self._hour == 0 or self._hour == 12:
72            standardTime += "12:"
73         else:
74            standardTime += "%d:" % ( self._hour % 12 )
75
76         standardTime += "%.2d:%.2d" % ( self._minute, self._second )
77
78         if self._hour < 12:
79            standardTime += " AM"
80         else:
81            standardTime += " PM"
82
83         print standardTime,
```

Fig. 7.7 Access methods defined for class **`Time`**. (Part 3 of 3.)

Notice that the constructor creates attributes with single leading underscores (_) in lines 10–12. Attribute names that begin with a single underscore have no special meaning in the syntax of the Python language itself. However, the single leading underscore is a convention among Python programmers who use the class. When a class author creates an attribute with a single leading underscore, the author does not want users of the class to access the attribute directly. If a program requires access to the attributes, the class author provides some other means for doing so. In this case, we provide access methods through which clients should manipulate the data.

Good Programming Practice 7.6

An attribute with a single leading underscore conveys information about a class's interface. Clients of a class that defines such attributes should access and modify the attributes' values only through the access methods that the class provides. Failing to do so often causes unexpected errors to occur during program execution.

Software Engineering Observation 7.6

Python's classes and modularity encourage programs to be implementation independent. When the implementation of a class used by implementation-independent code changes, that code need not be modified.

Method **`setTime`** (lines 14–19) is the *set* method that clients should use to set all values in an object's time. This method receives as arguments values for attributes **`_hour`**, **`_minute`** and **`_second`**. Methods **`setHour`** (lines 21–27), **`setMinute`** (lines 29–35) and **`setSecond`** (lines 37–43) are *set* methods for the individual attributes. These methods provide more flexibility to clients that modify the time.

Software Engineering Observation 7.7

Not all methods need to serve as part of a class's interface. Some methods serve as utility methods to other methods of the class and are not intended to be used by clients of the class.

Common Programming Error 7.6

When inside a method, forgetting to use the object reference (often called **`self`***) to access another method defined by the object's class is either a fatal runtime error or a logic error. The logic error occurs when the global namespace contains a function with the same name as one of the class's methods. In this case, forgetting to access the method name through the object reference actually calls the global function.*

The comparison expressions in lines 24, 32 and 40 demonstrate Python's comparison "chaining" syntax that enables programmers to write comparison expressions in familiar, arithmetic terms. Chained comparison expressions can be re-written with syntax familiar from other languages, using an appropriate **`and`** or **`or`** operation. For example, the statement in line 24 also could be written as

```
hour >= 0 and hour < 24
```

Performance Tip 7.1

Chained comparison expressions can be more efficient than their non-chained equivalents, because each term in the chained comparison expression is evaluated only once.

Method **`setHour`** (lines 21–27) changes an object's **`_hour`** attribute. The method checks whether the value passed as a parameter is in the range 0–23, inclusive. If the hour is valid, the method updates attribute **`_hour`** with the new value. Otherwise, the method *raises an exception*, to indicate that the client has attempted to place the object's data in an inconsistent state. An *exception* is a Python object that indicates a special event (most often an error) has occurred. For example, when a program attempts to access a nonexistent dictionary key, Python raises an exception.

When an exception is raised a program either can *catch* the exception and *handle* it; or the exception can go uncaught, in which case the program prints an error message and terminates immediately. Catching and handling an exception enables a program to recognize and potentially fix errors that might otherwise cause a program to terminate. For example, a client that uses class **`Time`** can catch an exception and detect that the program has attempted to place data in an inconsistent state (i.e., set an invalid time). Catching and handling exceptions is a broad topic that we discuss in detail in Chapter 12, Exception Handling. For now, we discuss only how to raise an exception to indicate invalid data assignments and prevent data corruption.

The statement in line 27 uses keyword ***`raise`*** to raise an exception. The keyword **`raise`** is followed by the name of the exception, a comma and a value that the exception object stores as an attribute. When Python executes a **`raise`** statement, an exception is raised; if the exception is not caught, Python prints an error message that contains the name of the exception and the exception's attribute value, as shown in Fig. 7.8.

The remaining methods—**`setMinute`** (lines 29–35) and **`setSecond`** (lines 37–43) change attributes **`_minute`** and **`_second`**, respectively. Each method ensures that the values remain in the range 0–59, inclusive. If the values are invalid, the methods raise exceptions and specify appropriate error-message arguments.

```
Python 2.2b2 (#26, Nov. 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>>
>>> raise ValueError, "This is an error message"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: This is an error message
```

Fig. 7.8 Raising an exception.

Lines 45–58 contain the *get* methods for class **`Time`**. Clients use these methods (**`getHour`**, **`getMinute`** and **`getSecond`**) to retrieve the values of an attributes **`_hour`**, **`_minute`** and **`_second`**, respectively. The remainder of the class definition does not differ from the previous definition we presented.

Software Engineering Observation 7.8

If a class provides access methods for its data, clients should use only access methods to retrieve/modify data. This "agreement" between class and client helps maintain data in a consistent state.

Software Engineering Observation 7.9

The class designer need not provide set *or* get *methods for each data item; these capabilities should be provided only when appropriate. If the service is appropriate for clients, that service should be provided in the class's interface.*

Software Engineering Observation 7.10

Every method that modifies the data of an object should ensure that the data remains in a consistent state.

Figure 7.9 contains a *driver* for modified class **`Time`**. A driver is a program that tests a class's interface. Lines 4–6 **`import`** class **`Time`** from module **`Time2`** and create an object of the class. Lines 9–12 call methods **`printMilitary`** and **`printStandard`** to display the initial time values of the object.

```
1   # Fig. 7.9: fig07_09.py
2   # Driver to test class TimeControl.
3
4   from Time2 import Time
5
6   time1 = Time()
7
8   # print initial time
9   print "The initial military time is",
10  time1.printMilitary()
11  print "\nThe initial standard time is",
12  time1.printStandard()
```

Fig. 7.9 Access methods called to change data. (Part 1 of 2.)

```
13
14  # change time
15  time1.setTime( 13, 27, 6 )
16  print "\n\nMilitary time after setTime is",
17  time1.printMilitary()
18  print "\nStandard time after setTime is",
19  time1.printStandard()
20
21  time1.setHour( 4 )
22  time1.setMinute( 3 )
23  time1.setSecond( 34 )
24  print "\n\nMilitary time after setHour, setMinute, setSecond is",
25  time1.printMilitary()
26  print "\nStandard time after setHour, setMinute, setSecond is",
27  time1.printStandard()
```

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

Military time after setHour, setMinute, setSecond is 04:03:34
Standard time after setHour, setMinute, setSecond is 4:03:34 AM
```

Fig. 7.9 Access methods called to change data. (Part 2 of 2.)

Line 15 calls **`time1`**'s method **`setTime`**, passing values that correspond to 1:27:06 PM, to change the object's time values. Lines 16–19 call the appropriate methods to display the formatted times. The interactive session in Fig. 7.10 creates an object of class **`Time`** and calls method **`setTime`** to attempt to place the object's data in an inconsistent state. Each call to method **`setTime`** contains an invalid value, and each call results in an error message.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>>
>>> from Time2 import Time
>>> time1 = Time()
>>>
>>> time1.setHour( 30 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "Time2.py", line 27, in setHour
    raise ValueError, "Invalid hour value: %d" % hour
ValueError: Invalid hour value: 30
```

(continued top of next page)

Fig. 7.10 *set* method called with invalid values. (Part 1 of 2.)

(continued from previous page)

```
>>>
>>> time1.setMinute( 99 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "Time2.py", line 35, in setMinute
    raise ValueError, "Invalid minute value: %d" % minute
ValueError: Invalid minute value: 99
>>>
>>> time1.setSecond( -99 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "Time2.py", line 43, in setSecond
    raise ValueError, "Invalid second value: %d" % second
ValueError: Invalid second value: -99
```

Fig. 7.10 *set* method called with invalid values. (Part 2 of 2.)

7.4.2 Private Attributes

In programming languages such as C++ and Java, a class may state explicitly which attributes or methods may be accessed by clients of the class. These attributes or methods are said to be *public*. Attributes and methods that may not be accessed by clients of the class are said to be *private*.

In Python, an object's attributes may always be accessed—there is no way to prevent other code from accessing the data. However, Python does provide a way to prevent indiscriminate access to data. Suppose we want to create an object of class **Time** and to prevent the following assignment statement

```
time1.hour = 25
```

To prevent such access, we prefix the name of the attribute with two underscore characters (__). When Python encounters an attribute name that begins with two underscores, the interpreter performs *name mangling* on the attribute, to prevent indiscriminate access to the data. Name mangling changes the name of an attribute by including information about the class to which the attribute belongs. For example, if the **Time** constructor contained the line

```
self.__hour = 0
```

Python creates an attribute called **_Time__hour**, instead of an attribute called **__hour**. Figure 7.11 contains an example in which we define a class **PrivateClass** that contains one public attribute **publicData** (line 10) and one private attribute **__privateData** (line 11). The interactive session that follows (Fig. 7.12) demonstrates how to access an object's data.

First, in Fig. 7.12, we import the class from module **Private** and create an object called **private**. The statement

```
print private.publicData
```

```python
# Fig. 7.11: Private.py
# Class with private data members.

class PrivateClass:
   """Class that contains public and private data"""

   def __init__( self ):
      """Private class, contains public and private data members"""

      self.publicData = "public"       # public data member
      self.__privateData = "private"   # private data member
```

Fig. 7.11 Class **PrivateClass** with private data.

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>>
>>> from Private import PrivateClass
>>> private = PrivateClass()
>>> print private.publicData
public
>>> print private.__privateData
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: PrivateClass instance has no attribute
'__privateData'
>>>
>>> print private._PrivateClass__privateData
private
>>> private._PrivateClass__privateData = "modified"
>>> print private._PrivateClass__privateData
modified
```

Fig. 7.12 Private data accessed.

behaves as expected—Python prints the value of the public attribute. When we write the statement

```python
print private.__privateData
```

Python prints an error message which explains that class **PrivateClass** does not contain an attribute called **__privateData**. We prefixed our attribute name with double underscores, so Python changed the name of the attribute in the class definition.

However, we can still access the data, because we know that Python renames attribute **__privateData** to attribute **_PrivateClass__privateData**. Therefore, the line

```python
print private._PrivateClass__privateData
```

successfully prints the value assigned to the private attribute. The final two statements in the session demonstrate that private data may be modified in the same way as public data.

However, accessing and modifying private attributes in this manner violates the data encapsulation the class author intended. A client should never perform such a manipulation, but instead should use any access methods the class provides.

Software Engineering Observation 7.11

Make private any data that the client should not access.

Python programmers use private attributes for different reasons. Some programmers use private attributes to avoid common scoping problems that may arise in inheritance hierarchies. [*Note*: We discuss inheritance in Chapter 9, Object-Oriented Programming: Inheritance.] Other programmers use private attributes for data or methods the client should never access. These attributes or methods are essential to the inner workings of the class, but are not part of the class's interface. For example, a class author might designate a utility method by prepending the method name with two underscores. In this chapter, we use private attributes to demonstrate access methods and to introduce a basic data integrity technique. In the next chapter, we discuss other ways to ensure data integrity. The techniques we discuss in the next chapter allow programmers to use public access syntax but also to take advantage of the data integrity provided by access methods. This practice enables programmers to add data integrity to a project as the project grows and matures, without having to change the interface on which the project's clients have come to rely.

7.5 Using Default Arguments With Constructors

Thus far, the client has supplied all the values that the constructor for class **`Time`** needed to initialize a new object. However, constructors can define default arguments that specify initial values for an object's attributes, if the client does not specify an argument at construction time. Constructors also can define keyword arguments that enable the client to specify values for only certain, named arguments. Figure 7.13—**`Time3.py`**—defines a modified version of class **`Time`** that redefines the **`Time`** constructor to include the default value 0 for each argument. Providing a default constructor guarantees that objects will be initialized to consistent states, even if no values are provided in constructor calls. Programmer-supplied constructors that default all their arguments (or explicitly require no arguments) are also called *default constructors* (i.e., constructors that can be invoked with no arguments.)

```
# Fig: 7.13: Time3.py
# Class Time with default constructor.

class Time:
   """Class Time with default constructor"""

   def __init__( self, hour = 0, minute = 0, second = 0 ):
      """Time constructor initializes each data member to zero"""

      self.setTime( hour, minute, second )

```

Fig. 7.13 Default constructor defined for class **`Time`**. (Part 1 of 3.)

```
   def setTime( self, hour, minute, second ):
      """Set values of hour, minute, and second"""

      self.setHour( hour )
      self.setMinute( minute )
      self.setSecond( second )

   def setHour( self, hour ):
      """Set hour value"""

      if 0 <= hour < 24:
         self.__hour = hour
      else:
         raise ValueError, "Invalid hour value: %d" % hour

   def setMinute( self, minute ):
      """Set minute value"""

      if 0 <= minute < 60:
         self.__minute = minute
      else:
         raise ValueError, "Invalid minute value: %d" % minute

   def setSecond( self, second ):
      """Set second value"""

      if 0 <= second < 60:
         self.__second = second
      else:
         raise ValueError, "Invalid second value: %d" % second

   def getHour( self ):
      """Get hour value"""

      return self.__hour

   def getMinute( self ):
      """Get minute value"""

      return self.__minute

   def getSecond( self ):
      """Get second value"""

      return self.__second

   def printMilitary( self ):
      """Prints Time object in military format"""

      print "%.2d:%.2d:%.2d" % \
         ( self.__hour, self.__minute, self.__second ),

```

Fig. 7.13 Default constructor defined for class **`Time`**. (Part 2 of 3.)

```
    def printStandard( self ):
       """Prints Time object in standard format"""

       standardTime = ""

       if self.__hour == 0 or self.__hour == 12:
          standardTime += "12:"
       else:
          standardTime += "%d:" % ( self.__hour % 12 )

      standardTime += "%.2d:%.2d" % ( self.__minute, self.__second )

       if self.__hour < 12:
          standardTime += " AM"
       else:
          standardTime += " PM"

       print standardTime,
```

Fig. 7.13 Default constructor defined for class **Time**. (Part 3 of 3.)

In this example, the constructor invokes method **setTime** with the values passed to the constructor (or the default values). The class uses private attributes to store data. As with the previous definition of **Time**, **setTime** uses the class's other methods, which ensure that the value supplied for **__hour** is in the range 0–23 and that the values for **__minute** and **__second** are each in the range 0–59. If a value is out of range, the appropriate method raises an exception (this is an example of ensuring that a data member remains in a consistent state).

The **Time** constructor could have included the same statements as method **setTime**. This may be slightly more efficient because the extra call to **setTime** is eliminated. Coding the **Time** constructor and method **setTime** identically, however, makes maintaining this class more difficult. If the implementation of method **setTime** changes, the implementation of the **Time** constructor should change accordingly. Instead, any changes to the implementation of **setTime** need to be made only once, because the **Time** constructor calls **setTime** directly. This reduces the likelihood of a programming error when altering the implementation.

Software Engineering Observation 7.12

If a method of a class already provides all or part of the functionality required by a constructor (or other method) of the class, call that method from the constructor (or other method). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

Figure 7.14 initializes four objects of class **Time** (defined in Fig. 7.13)—one with all three arguments defaulted in the constructor call, one with one argument specified, one with two arguments specified and one with three arguments specified. The values of each object's attributes after initialization are displayed by calling **printTimeValues** (lines 6–10).

If no constructor is defined for a class, the interpreter creates a default constructor (i.e., one that can be called with no arguments). However, the constructor that Python provides

does not perform any initialization, so, when an object is created, the object is not guaranteed to be in a consistent state.

```
# Fig. 7.14: fig07_14.py
# Demonstrating default constructor method for class Time.

from Time3 import Time

def printTimeValues( timeToPrint ):
    timeToPrint.printMilitary()
    print
    timeToPrint.printStandard()
    print

time1 = Time()              # all default
time2 = Time( 2 )           # minute, second default
time3 = Time( 21, 34 )      # second default
time4 = Time( 12, 25, 42 ) # all specified

print "Constructed with:"

print "\nall arguments defaulted:"
printTimeValues( time1 )

print "\nhour specified; minute and second defaulted:"
printTimeValues( time2 )

print "\nhour and minute specified; second defaulted:"
printTimeValues( time3 )

print "\nhour, minute and second specified:"
printTimeValues( time4 )
```

```
Constructed with:

all arguments defaulted:
00:00:00
12:00:00 AM

hour specified; minute and second defaulted:
02:00:00
2:00:00 AM

hour and minute specified; second defaulted:
21:34:00
9:34:00 PM

hour, minute and second specified:
12:25:42
12:25:42 PM
```

Fig. 7.14 Objects created with default constructor.

7.6 Destructors

A constructor is a method that initializes a newly created object. Conversely, a *destructor* executes when an object is destroyed (e.g., after no more references to the object exist). A class can define a special method called ***`__del__`*** that executes when the last reference to an object is deleted or goes out of scope[1]. The method itself does not actually destroy the object—it performs *termination housekeeping* before the interpreter reclaims the object's memory, so that memory may be reused. A destructor normally specifies no parameters other than **`self`** and returns **`None`**.

We have not defined method **`__del__`** for the classes presented to this point. In programming languages such as C++, destructors often allocate and recycle memory. Python handles most of these issues for the programmer, so **`__del__`** normally is not included in the class definition. Occasionally, a class defines **`__del__`** to close a network or a database connection before destroying an object. We discuss these issues throughout the text, as appropriate. In the next section, we define method **`__del__`** for a class, to maintain a count of all objects of the class that have been created.

7.7 Class Attributes

Each object of a class has copies of all the attributes created in the constructor. In certain cases, only one copy of an attribute should be shared by all objects of a class. A *class attribute* is used for this reason. A class attribute represents "class-wide" information (i.e., a property of the class, not of a specific object of the class).

We now consider a video-game example to justify the need for class-wide data. Suppose that we have a video game with **`Martian`**s and other space creatures. Each **`Martian`** tends to be brave and willing to attack other space creatures when the **`Martian`** is aware that there are at least four other **`Martian`**s present. If there are fewer than five **`Martian`**s present, each **`Martian`** becomes cowardly. For this reason, each **`Martian`** must know the **`martianCount`**. We could endow each object of class **`Martian`** with **`martianCount`** as an attribute. However, if we do this, then every **`Martian`** would have a separate copy of the attribute, and, each time we create a **`Martian`**, we would have to update attribute **`martianCount`** in every **`Martian`**. The redundant copies waste space, and updating those copies is time-consuming. Instead, we create **`martianCount`** as a class attribute so that **`martianCount`** is class-wide data. Each **`Martian`** can see the **`martianCount`** as if it were an attribute of that **`Martian`**, but Python maintains only one copy of the **`martianCount`** attribute to save space. This technique also saves time; because there is only one copy, we do not have to increment separate copies of **`martianCount`** for each object of class **`Martian`**.

Performance Tip 7.2

When a single copy of the data will suffice, use class attributes to save storage.

1. Actually, there are some cases in which **`__del__`** does not execute immediately after the last reference to an object is deleted. However, in most cases, it is safe to assume that the method executes when expected. See **`www.python.org/doc/current/ref/customization.html`** for more information.

Although class attributes may seem like global variables, each class attribute resides in the namespace of the class in which it is created. Class attributes should be initialized once (and only once) in the class definition. A class's class attributes can be accessed through any object of that class. A class's class attributes also exist even when no object of that class exists. To access a class attribute when no object of the class exists, prefix the class name, followed by a period, to the name of the attribute.

Software Engineering Observation 7.13

A class's class attributes can be used even if no objects of that class have been instantiated.

Class **Employee** (Fig. 7.15) demonstrates how to define a class attribute that maintains a count of the number of objects of the class that have been instantiated. The class attribute **count** is initialized to 0 in the class definition (line 7). Notice that the creation of class attribute **count** appears in the body of the class definition, not inside a method. The statement has the effect of defining a new variable named **count**, with value 0, and adding that variable to class **Employee**'s namespace.

```
1  # Fig. 7.15: EmployeeWithClassAttribute.py
2  # Class Employee with class attribute count.
3
4  class Employee:
5     """Represents an employee"""
6
7     count = 0       # class attribute
8
9     def __init__( self, first, last ):
10       """Initializes firstName, lastName and increments count"""
11
12       self.firstName = first
13       self.lastName = last
14
15       Employee.count += 1    # increment class attribute
16
17       print "Employee constructor for %s, %s" \
18          % ( self.lastName, self.firstName )
19
20    def __del__( self ):
21       """Decrements count and prints message"""
22
23       Employee.count -= 1    # decrement class attribute
24
25       print "Employee destructor for %s, %s" \
26          % ( self.lastName, self.firstName )
```

Fig. 7.15 Class attributes—class **Employee**.

```
# Fig. 7.16: fig07_16.py
# Demonstrating class attribute access.

```

Fig. 7.16 Class attributes—**fig07_16.py**. (Part 1 of 2.)

```
4  from EmployeeWithClassAttribute import Employee
5
6  print "Number of employees before instantiation is", \
7     Employee.count
8
9  # create two Employee objects
10 employee1 = Employee( "Susan", "Baker" )
11 employee2 = Employee( "Robert", "Jones" )
12 employee3 = employee1
13
14 print "Number of employees after instantiation is", \
15    employee1.count
16
17 # explicitly delete employee objects by removing references
18 del employee1
19 del employee2
20 del employee3
21
22 print "Number of employees after deletion is", \
23    Employee.count
```

```
Number of employees before instantiation is 0
Employee constructor for Baker, Susan
Employee constructor for Jones, Robert
Number of employees after instantiation is 2
Employee destructor for Jones, Robert
Employee destructor for Baker, Susan
Number of employees after deletion is 0
```

Fig. 7.16 Class attributes—**`fig07_16.py`**. (Part 2 of 2.)

Figure 7.16 access **`Employee`**'s class attribute. Class attribute **`count`** maintains a count of the number of existing objects of class **`Employee`** and can be accessed whether or not objects of class **`Employee`** exist. If no objects of the class exist, a program can reference **`count`** through the class name (line 7). Lines 10–11 create two **`Employee`** objects. When each **`Employee`** object is created, its constructor is called. In the output, notice that creating identifier **`employee3`** (line 12) does not create a new object of class **`Employee`** and therefore does not call **`Employee`**'s constructor. The statement simply binds a new name to the object created in line 18, so that **`employee3`** and **`employee1`** refer to the same object. Lines 18–20 use keyword **`del`** to delete all references to the two **`Employee`** objects. Method **`__del__`** for the object created in line 10 does not execute until the last reference to that object is deleted in line 20.

7.8 Composition: Object References as Members of Classes

Until now, we have defined classes whose objects have attributes of basic types. Sometimes, a programmer needs objects whose attributes are themselves references to objects of other classes. For example, an object of class **`AlarmClock`** needs to know when it is supposed to sound its alarm, so why not include an object of class **`Time`** as a member of the object of class **`AlarmClock`**? Such a capability is called *composition*.

Software Engineering Observation 7.14

One form of software reusability is composition, in which a class has references to objects of other classes as members.

Software Engineering Observation 7.15

If a class has as a member a reference to an object of another class, making that member object publicly accessible does not violate the encapsulation and hiding of that member object's private members.

Figure 7.17 uses a class **`Date`** (Fig. 7.17) a modified class **`Employee`** (Fig. 7.18) and to demonstrate references to objects as members of other objects. Class **`Employee`** contains attributes **`firstName`**, **`lastName`**, **`birthDate`** and **`hireDate`**. Attributes **`birthDate`** and **`hireDate`** are objects of class **`Date`**, which contains attributes **`month`**, **`day`** and **`year`**. The program (Fig. 7.19) instantiates an object of class **`Employee`** and initializes and displays its attributes.

```python
# Fig. 7.17: Date.py
# Definition of class Date.

class Date:
   """Class that represents dates"""

   # class attribute lists number of days in each month
   daysPerMonth = [
      0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]

   def __init__( self, month, day, year ):
      """Constructor for class Date"""

      if 0 < month <= 12:   # validate month
         self.month = month
      else:
         raise ValueError, "Invalid value for month: %d" % month

      if year >= 0:         # validate year
         self.year = year
      else:
         raise ValueError, "Invalid value for year: %y" % year

      self.day = self.checkDay( day )   # validate day

      print "Date constructor:",
      self.display()

   def __del__( self ):
      """Prints message when called"""

      print "Date object about to be destroyed:",
      self.display()

```

Fig. 7.17 Member objects—**`Date.py`**. (Part 1 of 2.)

```
    def display( self ):
       """Prints Date information"""

       print "%d/%d/%d" % ( self.month, self.day, self.year )

    def checkDay( self, testDay ):
       """Validates day of the month"""

       # validate day, test for leap year
       if 0 < testDay <= Date.daysPerMonth[ self.month ]:
          return testDay
       elif self.month == 2 and testDay == 29 and \
          ( self.year % 400 == 0 or
            self.year % 100 != 0 and self.year % 4 == 0 ):
          return testDay
       else:
          raise ValueError, "Invalid day: %d for month: %d" % \
             ( testDay, self.month )
```

Fig. 7.17 Member objects—**Date.py**. (Part 2 of 2.)

In Fig. 7.18, the **Employee** constructor (lines 9–20) takes nine arguments—**self**, **firstName**, **lastName**, **birthMonth**, **birthDay**, **birthYear**, **hireMonth**, **hireDay** and **hireYear**—and creates objects of class **Date** from the last six arguments. Arguments **birthMonth**, **birthDay** and **birthYear** are passed to object **birthDate**'s constructor, and arguments **hireMonth**, **hireDay** and **hireYear** are passed to object **hireDate**'s constructor. Class **Date** and class **Employee** each define method **__del__** to print a message when an object of class **Date** or an object of class **Employee** is destroyed, respectively.

```
1  # Fig. 7.18: EmployeeComposition.py
2  # Definition of Employee class with composite members.
3
4  from Date import Date
5
6  class Employee:
7     """Employee class with Date attributes"""
8
9     def __init__( self, firstName, lastName, birthMonth,
10       birthDay, birthYear, hireMonth, hireDay, hireYear ):
11       """Constructor for class Employee"""
12
13       self.birthDate = Date( birthMonth, birthDay, birthYear )
14       self.hireDate = Date( hireMonth, hireDay, hireYear )
15
16       self.lastName = lastName
17       self.firstName = firstName
18
19       print "Employee constructor: %s, %s" \
20          % ( self.lastName, self.firstName )
```

Fig. 7.18 Member objects—**EmployeeComposition.py**. (Part 1 of 2.)

```

   def __del__( self ):
      """Called before Employee destruction"""

      print "Employee object about to be destroyed: %s, %s" \
         % ( self.lastName, self.firstName )

   def display( self ):
      """Prints employee information"""

      print "%s, %s" % ( self.lastName, self.firstName )
      print "Hired:",
      self.hireDate.display()
      print "Birth date:",
      self.birthDate.display()
```

Fig. 7.18 Member objects—`EmployeeComposition.py`. (Part 2 of 2.)

```
# Fig. 7.19: fig07_19.py
# Demonstrating composition: an object with member objects.

from Date import Date
from EmployeeComposition import Employee

employee = Employee( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 )
print

employee.display()
print
```

```
Date constructor: 7/24/1949
Date constructor: 3/12/1988
Employee constructor: Jones, Bob

Jones, Bob
Hired: 3/12/1988
Birth date: 7/24/1949

Employee object about to be destroyed: Jones, Bob
Date object about to be destroyed: 3/12/1988
Date object about to be destroyed: 7/24/1949
```

Fig. 7.19 Member objects—`fig07_19.py`.

7.9 Data Abstraction and Information Hiding

As we pointed out at the beginning of this chapter, classes normally hide the details of their implementation from their clients. This is called *information hiding.* As an example of information hiding, let us consider a data structure called a *stack*.

Students can think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it is always placed at the top (referred to as *pushing* the dish onto the stack). Similarly, when a dish is removed from the pile, it is always removed from the top (referred to as *pop-*

ping the dish off the stack). Stacks are known as *last-in, first-out (LIFO) data structures*—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

Stacks can easily be implemented with lists, and in fact, Python lists contain methods that programers can use to make lists "act" like stacks. (We also implement our own class **`Stack`** in Chapter 22, Data Structures.) A client of a stack class need not be concerned with the stack's implementation. The client knows only that when data items are placed in the stack, these items will be recalled in last-in, first-out order. The client cares about *what* functionality a stack offers, but not about *how* that functionality is implemented. This concept is referred to as *data abstraction*. Although programmers might know the details of a class's implementation, they should not write code that depends on these details. This enables a particular class (such as one that implements a stack and its operations, *push* and *pop*) to be replaced with another version without affecting the rest of the system. As long as the services of the class do not change (i.e., every method still has the same name, returns the same type of value and defines the same parameter list in the new class definition), the rest of the system is not affected.

The job of a high-level language is to create a view convenient for programmers to use. There is no single accepted standard view—that is one reason why there are so many programming languages. Object-oriented programming in Python presents yet another view.

Most programming languages emphasize actions. In these languages, data exists to support the actions that programs must take. Data is "less interesting" than actions. Data is "crude." Only a few built-in data types exist, and it is difficult for programmers to create their own data types. The object-oriented style of programming in Python elevates the importance of data. The primary activities of object-oriented programming in Python is the creation of data types (i.e., classes) and the expression of the interactions among objects of those data types. To create languages that emphasize data, the programming-languages community needed to formalize some notions about data. The formalization we consider here is the notion of *abstract data types (ADTs)*. ADTs receive as much attention today as structured programming did decades earlier. ADTs, however, do not replace structured programming. Rather, they provide an additional formalization to improve the program-development process.

Consider the built-in integer type, which most people would associate with an integer in mathematics. Rather, the integer type is an abstract representation of an integer. Unlike mathematical integers, computer integers are fixed in size. For example, the integer type on some computers is limited approximately to the range –2 billion to +2 billion. If the result of a calculation falls outside this range, an error occurs, and the computer responds in some machine-dependent manner. It might, for example, "quietly" produce an incorrect result. Mathematical integers do not have this problem. Therefore, the notion of a computer integer is only an approximation of the notion of a real-world integer. The same is true of the floating-point type and other built-in types.

We have taken the notion of the integer type for granted until this point, but we now consider it from a new perspective. Types like integer, floating-points, strings and others are all examples of abstract data types. These types are representations of real-world notions to some satisfactory level of precision within a computer system.

An ADT actually captures two notions: A *data representation* and the *operations* that can be performed on that data. For example, in Python, an integer contains an integer value (data) and provides addition, subtraction, multiplication, division and modulus operations;

however, division by zero is undefined. Python programmers use classes to implement abstract data types.

Software Engineering Observation 7.16

Programmers can create types through the use of the class mechanism. These new types can be designed so that they are as convenient to use as the built-in types. This marks Python as an extensible language. *Although the language is easy to extend via new types, the programmer cannot alter the base language itself.*

Another abstract data type we discuss is a *queue*, which is similar to a "waiting line." Computer systems use many queues internally. A queue offers well-understood behavior to its clients: Clients place items in a queue one at a time via an *enqueue* operation, then get those items back one at a time via a *dequeue* operation. A queue returns items in *first-in, first-out (FIFO)* order, which means that the first item inserted in a queue is the first item removed. Conceptually, a queue can become infinitely long, but real queues are finite.

The queue hides an internal data representation that keeps track of the items currently waiting in line, and it offers a set of operations to its clients (*enqueue* and *dequeue*). The clients are not concerned about the implementation of the queue—clients simply depend upon the queue to operate "as advertised." When a client enqueues an item, the queue should accept that item and place it in some kind of internal FIFO data structure. Similarly, when the client wants the next item from the front of the queue, the queue should remove the item from its internal representation and deliver the item in FIFO order (i.e., the item that has been in the queue the longest should be the next one returned by the next dequeue operation).

The queue ADT guarantees the integrity of its internal data structure. Clients cannot manipulate this data structure directly—only the queue ADT has access to its internal data. Clients are able to perform only allowable operations on the data representation; the ADT rejects operations that its interface does not provide. This could mean issuing an error message, terminating execution, raising an exception (as discussed in Chapter 12, Exception Handling) or simply ignoring the operation request.

7.10 Software Reusability

Python programmers concentrate both on crafting new classes and on reusing classes from the standard library, which contains hundreds of predefined classes. Developers construct software by combining programmer-defined classes with well-defined, carefully tested, well-documented, portable and widely available standard library classes. This kind of software reusability speeds the development of powerful, high-quality software. *Rapid applications development (RAD)* is of great interest today.

The standard library enables Python developers to build applications faster by reusing preexisting, extensively tested classes. In addition to reducing development time, standard library classes also improve programmers' abilities to debug and maintain applications, because proven software components are being used. For programmers to take advantage of the standard library's classes, they must familiarize themselves with the standard library's rich set of capabilities.

In this chapter, we discussed how to define a class and to create objects of the class. When a new object is created, the class constructor initializes the new object's attributes. We discussed several ways to initialize and modify attributes—default constructors, *set* methods and raising exceptions for invalid attribute values. We also discussed data integ-

rity, how all object attributes may be accessed directly by the client, how the single leading underscore (_) indicates that clients should not access attributes and how the double leading underscore (__) mangles an attribute's name to prevent casual attribute access. Python's direct attribute access encourages rapid application development and facilitates dynamic introspection; however, direct access is often insufficient for large-scale software projects. In the next chapter, we discuss how class authors can ensure data integrity, while still taking advantage of direct access syntax. This data integrity functionality can be added to the class without changing the interface the client uses to access an object's data. This promotes both the safe, modular programming techniques and rapid development practices that Python programmers desire.

SUMMARY

- Object-oriented programming (OOP) encapsulates (i.e., wraps) data (attributes) and functions (behaviors) into components called classes. The data and functions of a class are intimately tied together.
- A class is like a blueprint. Using a blueprint, a builder can build a house. Using a class, a programmer can create an object (also called an instance).
- Classes have a property called *i*nformation hiding. Although objects may know how to communicate with one another across well-defined interfaces, one object normally should not be allowed to know how another object is implemented—implementation details are hidden within the objects themselves.
- In procedural programming, the unit of programming is the function. In object-oriented programming, the unit of programming is the class from which objects eventually are instantiated.
- Procedural programmers concentrate on writing functions. The verbs in a system specification help the procedural programmer determine the set of functions that will work together to implement the system.
- Object-oriented programmers concentrate on creating their own user-defined types, called classes. The nouns in a system specification help the object-oriented programmer determine the set of classes that will be used to create the objects that will work together to implement the system.
- Classes simplify programming because the clients need to be concerned only with the operations encapsulated or embedded in the object—the object interface.
- Keyword **class** begins a class definition. The keyword is followed by the name of the class, which is followed by a colon (**:**). The line that contains keyword **class** and the class name is called the class's header.
- The body of the class is an indented code block that contains methods and attributes that belong to the class.
- A class's optional documentation string describes the class. If a class contains a documentation string, the string must appear in the line or lines following the class header.
- Method __**init**__ is the constructor method of a class. A constructor is a special method that executes each time an object of a class is created. The constructor initializes the attributes of the object and returns **None**.
- All methods, including constructors, must specify at least one parameter—the object reference. This parameter represents the object of the class for which the method is called. Methods must use the object reference to access attributes and other methods that belong to the class.
- By convention, the object reference argument is called **self**.
- Each object has its own namespace that contains the object's methods and attributes. The class's constructor starts with an empty object (**self**) and adds attributes to the object's namespace.

- Once a class has been defined, programs can create objects of that class. Programmers can create objects as necessary. This is one reason why Python is said to be an *extensible language*.
- One of the fundamental principles of good software engineering is that a client should not need to know how a class is implemented to use that class. Python's use of modules facilitates this data abstraction—a program can import a class definition and use the class without knowing how the class is implemented.
- To create an object of a class, simply "call" the class name as if it were a function. This call invokes the constructor for the class.
- Classes and objects of classes both have special attributes that can be manipulated. These attributes, which Python creates when a class is defined or when an object of a class is created, provide information about the class or object of a class to which they belong.
- Directly accessing an object's data can leave the data in an inconsistent state.
- Most object-oriented programming languages allow an object to prevent its clients from accessing the object's data directly. However, in Python, the programmer uses attribute naming conventions to hide data from clients.
- Although a client can access an object's data directly (and perhaps cause the data to enter an inconsistent state), a programmer can design classes to encourage correct use. One technique is for the class to provide access methods through which the data of the class can be read and written in a carefully controlled manner.
- Predicate methods are read-only access methods that test the validity of a condition.
- When a class defines access methods, a client should access an object's attributes only through those access methods.
- Classes often provide methods that allow clients to *set* or *get* the values of attributes. Although these methods need not be called *set* and *get*, they often are. *Get* methods also are called "query" methods.
- A *get* method can control the formatting of the data. A *set* method can—and most likely should—scrutinize attempts to modify the value of the attribute. This ensures that the new value is appropriate for that data item.
- A *set* method may specify that an error message—called an exception—be raised to the client when the client attempts to assign an invalid value to an attribute.
- When a class author creates an attribute with a single leading underscore, the author does not want users of the class to access the attribute directly. If a program requires access to the attributes, the class author provides some other means for doing so.
- Python comparisons may be chained. The chaining syntax that enables programmers to write comparison expressions in familiar, arithmetic terms.
- When an exception is raised a program either can catch the exception and handle it; or the exception can go uncaught, in which case the program prints an error message and terminates immediately.
- The keyword **`raise`** is followed by the name of the exception, a comma and a value that the exception object stores as an attribute. When Python executes a **`raise`** statement, an exception is raised. If the exception is not caught, Python prints an error message that contains the name of the exception and the exception's attribute value.
- In programming languages such as C++ and Java, a class may state explicitly which attributes or methods may be accessed by clients of the class. These attributes or methods are said to be public. Attributes and methods that may not be accessed by clients of the class are said to be private.
- To prevent indiscriminate attribute access, prefix the name of the attribute with two underscore characters (__).

- When Python encounters an attribute name that begins with two underscores, the interpreter performs name mangling on the attribute, to prevent indiscriminate access to the data. Name mangling changes the name of an attribute by including information about the class to which the attribute belongs.
- Constructors can define default arguments that specify initial values for an object's attributes, if the client does not specify an argument at construction time.
- Constructors can define keyword arguments that enable the client to specify values for only certain, named arguments.
- Programmer-supplied constructors that default all their arguments (or explicitly require no arguments) are also called default constructors
- If no constructor is defined for a class, the interpreter creates a default constructor. However, the constructor that Python provides does not perform any initialization, so, when an object is created, the object is not guaranteed to be in a consistent state.
- A destructor executes when an object is destroyed (e.g., after no more references to the object exist).
- A class can define a special method called `__del__` that executes when the last reference to an object is deleted or goes out of scope. A destructor normally specifies no parameters other than `self` and returns `None`.
- A class attribute represents "class-wide" information (i.e., a property of the class, not of a specific object of the class).
- Although class attributes may seem like global variables, each class attribute resides in the namespace of the class in which it is created. Class attributes should be initialized once (and only once) in the class definition.
- A class's class attributes can be accessed through any object of that class. A class's class attributes also exist even when no object of that class exists. To access a class attribute when no object of the class exists, prefix the class name, followed by a period, to the name of the attribute.
- Sometimes, a programmer needs objects whose attributes are themselves references to objects of other classes. Such a capability is called composition.
- Stacks are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.
- Types like integer, floating-points, strings and others are all examples of abstract data types. These types are representations of real-world notions to some satisfactory level of precision within a computer system.
- An ADT actually captures two notions: A data representation and the operations that can be performed on that data. Python programmers use classes to implement abstract data types.
- A queue, is a "waiting line." A queue offers well-understood behavior to its clients: Clients place items in a queue one at a time via an enqueue operation, then get those items back one at a time via a dequeue operation.
- A queue returns items in first-in, first-out (FIFO) order, which means that the first item inserted in a queue is the first item removed.
- Python programmers concentrate both on crafting new classes and on reusing classes from the standard library. This kind of software reusability speeds the development of powerful, high-quality software.
- The standard library enables Python developers to build applications faster by reusing preexisting, extensively tested classes. In addition to reducing development time, standard library classes also improve programmers' abilities to deb

TERMINOLOGY

abstract data type (ADT)
access method
attribute
`__bases__` attribute of a class
behavior
built-in data type
class keyword
`__class__` attribute of an object
class body
class instance object
class library
class scope
composition
consistent state
constructor
container class
data abstraction
data member
data type
data validation
`__del__` method
del keyword
dequeue
destructor
`__dict__` attribute of a class
`__dict__` attribute of an object
`__doc__` attribute of a class
double underscore (__)
encapsulation
enqueue
extensible language
first in, first out (FIFO)
get access method
inconsistent state
information hiding
`__init__` method
instantiate
interface
last in, first out (LIFO)
member function
method
module
`__module__` attribute of a class
`__name__` attribute of a class
name mangling
object
object-oriented programming (OOP)
popping off a stack
predicate method
private
public
pushing onto a stack
queue
rapid application development (RAD)
reference
self
set access method
single underscore (_)
software reuse
stack
structured programming
termination housekeeping
user-defined type
utility method

SELF-REVIEW EXERCISES

7.1 Fill in the blanks in each of the following statements:

a) Object-oriented programming __________ data and functions into __________.
b) Method __________ is called the constructor.
c) Classes enable programmers to model objects that have __________ (represented as data members) and behaviors (represented as __________).
d) A class's methods are often referred to as __________ in other object-oriented programming languages.
e) A __________ method tests the truth or falsity of a condition.
f) A __________ is a variable shared by all objects of a class.
g) __________ are known as last-in, first-out data structures.
h) A user of an object is referred to as a __________.
i) Python performs name mangling on attributes that begin with __________ underscore(s).
j) Describing the functionality of a class independent of its implementation is called __________.

7.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Object-oriented programming languages do not use functions to perform actions.
b) The parameter **self** must be the first item in a method's argument list.
c) The class constructor returns an object of the class.
d) Programmer-defined and built-in modules are imported in the same way.
e) Constructors may specify keyword arguments and default arguments.
f) An attribute that begins with a single underscore is a private attribute.
g) The destructor is called when the keyword **del** is used on an object.
h) A shared class attribute should be initialized in the constructor.
i) When invoking an object's method, a program does not need to pass a value that corresponds to the object reference parameter.
j) Every class should have a **__del__** method to reclaim an object's memory.

ANSWERS TO SELF-REVIEW EXERCISES

7.1 a) encapsulates, classes. b) **__init__**. c) attributes, methods. d) member functions. e) predicate. f) class attribute. g) Stacks. h) client. i) two. j) data abstraction.

7.2 a) False. Object-oriented programmers use methods, or functions, as components of classes. b) False. The first parameter in a method's argument must be an object of the class, which is called **self** by convention. c) False. The class constructor initializes an object of the class and implicitly returns **None**. d) True. e) True. f) False. An attribute that begins with a single underscore conveys the convention that a client of a class should not access the attribute directly. g) False. A destructor executes when the last reference to an object is destroyed. h) False. A shared class attribute should be initialized exactly once, at class scope, outside the class's methods. i) True. j) False. The programmer is not required to write a **__del__** method for a class.

EXERCISES

7.3 Create a class called **Complex** for performing arithmetic with complex numbers. Write a driver program to test your class.

Complex numbers have the form

```
realPart + imaginaryPart * i
```

where ***i*** is

$$\sqrt{-1}$$

Use floating-point numbers to represent the data of the class. Provide a constructor that enables an object of this class to be initialized when it is created. The constructor should contain default values in case no initializers are provided. Provide methods for each of the following:

a) Adding two **ComplexNumber**s: The real parts are added to form the real part of the result, and the imaginary parts are added to form the imaginary part of the result.
b) Subtracting two **ComplexNumber**s: The real part of the right operand is subtracted from the real part of the left operand to form the real part of the result, and the imaginary part of the right operand is subtracted from the imaginary part of the left operand to form the imaginary part of the result.
c) Printing **ComplexNumber**s in the form **(a, b)**, where **a** is the real part and **b** is the imaginary part.

7.4 Create a class called **RationalNumber** for performing arithmetic with fractions. Write a driver program to test your class.

Use integer variables to represent the data of the class—the numerator and the denominator. Provide a constructor that enables an object of this class to be initialized when it is declared. The constructor should contain default values, in case no initializers are provided, and should store the fraction in reduced form (i.e., the fraction

$$\frac{2}{4}$$

would be stored in the object as 1 in the numerator and 2 in the denominator). Provide methods for each of the following:

a) Adding two **RationalNumber**s. The result should be stored in reduced form.
b) Subtracting two **RationalNumber**s. The result should be stored in reduced form.
c) Multiplying two **RationalNumber**s. The result should be stored in reduced form.
d) Dividing two **RationalNumber**s. The result should be stored in reduced form.
e) Printing **RationalNumber**s in the form **a/b**, where **a** is the numerator and **b** is the denominator.
f) Printing **RationalNumber**s in floating-point format.

7.5 Modify the **Time** class of Fig. 7.13 to include a **tick** method that increments the time stored in a **Time** object by one second. The **Time** object should always remain in a consistent state. Write a driver program that tests the **tick** method. Be sure to test the following cases:

a) Incrementing into the next minute.
b) Incrementing into the next hour.
c) Incrementing into the next day (i.e., 23:59:59 to 0:00:00).

7.6 Create a class **Rectangle**. The class has attributes **__length** and **__width**, each of which defaults to 1. It has methods that calculate the **perimeter** and the **area** of the rectangle. It has *set* and *get* methods for both **__length** and **__width**. The *set* methods should verify that **__length** and **__width** are each floating-point numbers larger than 0.0 and less than 20.0. Write a driver program to test the class.

7.7 Create a more sophisticated **Rectangle** class than the one you created in Exercise 7.6. This class stores only the *x-y* coordinates of the upper left-hand and lower right-hand corners of the rectangle. The constructor calls a *set* function that accepts two tuples of coordinates and verifies that each of these is in the first quadrant, with no single *x* or *y* coordinate larger than 20.0. Methods calculate the **length**, **width**, **perimeter** and **area**. The length is the larger of the two dimensions. Include a predicate method **isSquare** that determines whether the rectangle is a square. Write a driver program to test the class.

7.8 Create a class **TicTacToe** that will enable you to write a complete program to play the game of tic-tac-toe. The class contains a 3-by-3 double-subscripted list of letters. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place an **"X"** in the specified square; place an **"O"** wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won and whether the game is a draw. [*Note:* If you feel ambitious, modify your program so that the computer makes the moves for one of the players automatically. Also, allow the player to choose whether to go first or second.]

8

Customizing Classes

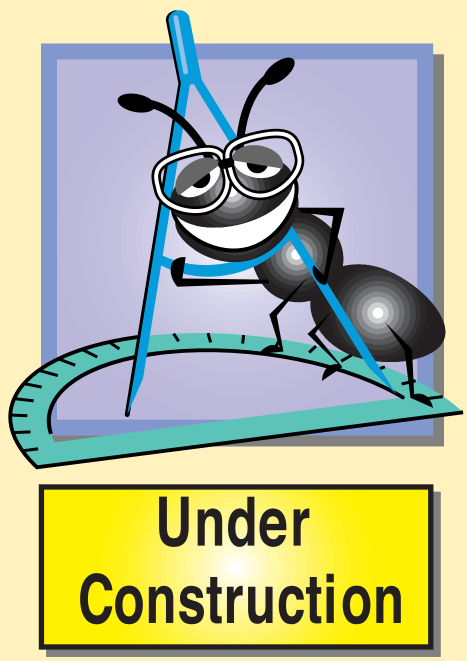
Objectives

- To understand how to write special methods that customize a class.
- To be able to represent an object as a string.
- To use special methods to customize attribute access.
- To understand how to redefine (overload) operators to work with new classes.
- To learn when to, and when not to, overload operators.
- To learn how to overload sequence operations.
- To learn how to overload mapping operations.
- To study interesting, customized classes.

The whole difference between construction and creation is exactly this: that a thing constructed can only be loved after it is constructed; but a thing created is loved before it exists.
Gilbert Keith Chesterton, Preface to Dickens, Pickwick Papers

The die is cast.
Julius Caesar

Our doctor would never really operate unless it was necessary. He was just that way. If he didn't need the money, he wouldn't lay a hand on you.
Herb Shriner



Outline

8.1 Introduction

In Chapter 7, we introduced the basics of Python classes and the notion of abstract data types (ADTs). We discussed how methods **`__init__`** and **`__del__`** execute when an object is created and destroyed, respectively. These methods are two examples of the many *special methods* that a class may define. A special method is a method that has a special meaning in Python; the Python interpreter calls one of an object's special methods when the client performs a certain operation on the object. For example, when a client creates an object of a class, Python invokes the **`__init__`** special method of that class.

A class author implements special methods to *customize* the behavior of the class. The purpose of customization is to provide the clients of a class with a simple notation for manipulating objects of the class. For example, in Chapter 7, manipulations on objects were accomplished by sending messages (in the form of method calls) to the objects. This method-call notation is cumbersome for certain kinds of classes, especially mathematical classes. For such classes, it would be nice to use Python's rich set of built-in operators and statements to manipulate objects. In this chapter, we show how to define special methods that enable Python's operators to work with objects—a process called *operator overloading*. It is straightforward and natural to extend Python with these new capabilities. Operator overloading also requires great care, because, when overloading is misused, it can make a program difficult to understand.

Operator **+** has multiple purposes in Python, for example, integer addition and string concatenation. This is an example of operator overloading. The Python language itself overloads operators **+** and *****, among others. These operators perform differently to suit the context in integer arithmetic, floating-point arithmetic, string manipulation and other operations.

Python enables the programmer to overload most operators to be sensitive to the context in which they are used. The interpreter takes the appropriate action based on the manner in which the operator is used. Some operators are overloaded frequently, especially operators like **+** and **-**. The job performed by overloaded operators also can be performed by explicit method calls, but operator notation is often clearer.

In this chapter, we discuss when to use operator overloading and when not to use it. We show how to overload operators, and we present complete programs using overloaded operators.

Customization provides other benefits, as well. A class may define special methods that cause an object of the class to behave like a list or like a dictionary. A class also may define special methods to control how a client accesses object attributes through the dot access operator. In this chapter, we introduce the appropriate special methods and create classes that implement them.

8.2 Customizing String Representation: Method `__str__`

Python is able to output the built-in data types with the **print** statement. What if a programmer wants to define a class whose objects also can be output with the **print** statement? A Python class can define special method **__*str*__**, to provide an informal (i.e., human-readable) string representation of an object of the class. If a client program of the class contains the statement

```
print objectOfClass
```

Python calls the object's **__str__** method and outputs the string returned by that method. Figure 8.1 demonstrates how to define special method **__str__** to handle data of a user-defined telephone number class called **PhoneNumber**. This program assumes telephone numbers are input correctly.

```
# Fig. 8.1: PhoneNumber.py
# Representation of phone number in USA format: (xxx) xxx-xxxx.

class PhoneNumber:
   """Simple class to represent phone number in USA format"""

   def __init__( self, number ):
      """Accepts string in form (xxx) xxx-xxxx"""

      self.areaCode = number[ 1:4 ]   # 3-digit area code
      self.exchange = number[ 6:9 ]   # 3-digit exchange
      self.line = number[ 10:14 ]   # 4-digit line

   def __str__( self ):
      """Informal string representation"""

      return "(%s) %s-%s" % \
         ( self.areaCode, self.exchange, self.line )

```

Fig. 8.1 String representation—special method `__str__`. (Part 1 of 2.)

```
20  def test():
21
22     # obtain phone number from user
23     newNumber = raw_input(
24        "Enter phone number in the form (123) 456-7890:\n" )
25
26     phone = PhoneNumber( newNumber )  # create PhoneNumber object
27     print "The phone number is:",
28     print phone  # invokes phone.__str__()
29
30  if __name__ == "__main__":
31     test()
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1234
The phone number is: (800) 555-1234
```

Fig. 8.1 String representation—special method `__str__`. (Part 2 of 2.)

Method `__init__` (lines 7–12) accepts a string in the form `"(xxx) xxx-xxxx"`, where each `x` in the string is a digit in the phone number. The method slices the string and stores the pieces of the phone number as attributes.

Method `__str__` (lines 14–18) is a special method that constructs and returns a string representation of an object of class `PhoneNumber`. When the interpreter encounters the statement

```
print phone
```

in line 28, the interpreter executes the statement

```
print phone.__str__()
```

When a program passes a `PhoneNumber` object to built-in function `str` or when a program uses a `PhoneNumber` object with the `%` string-formatting operator (e.g., `"%s" % phone`), Python also calls method `__str__`.



Common Programming Error 8.1

Returning a non-string value from method `__str__` is a fatal, runtime error.

Function `test`, (lines 20–28) requests a phone number from the user, creates a new `PhoneNumber` object, and prints the string representation of the object. Recall that when a module runs as a stand-alone program (i.e., the user invokes the Python interpreter on the module), Python assigns the value `"__main__"` to the namespace's name (stored in built-in variable `__name__`). Line 31 calls function `test`, if `PhoneNumber.py` is executed as a stand-alone program. This practice of defining a driver function and testing a module's namespace to execute the function is employed by many Python modules. The benefit of this practice is that a module author can define different behaviors for the module, based on the context in which the module is used. If another program imports the module, the value of `__name__` will be the module name (e.g., `"PhoneNumber"`), and the test function does

not execute. If the module is executed as a stand-alone program, the value of **`__name__`** is **`"__main__"`**, and the test function executes. In Chapters 10 and 11, we create graphical programs that use test functions to display the graphical components we define.

Good Programming Practice 8.1

Provide test functions for modules you create, when necessary. These functions help ensure that the module works correctly, and they provide additional information to clients of the class by demonstrating the ways in which a module's operations may be performed.

8.3 Customizing Attribute Access

In the previous chapter, we discussed two techniques for a client to access an object's attributes. The client can access the attributes directly (through the dot access operator), or the class author can give the attributes special names to signify that a client should access the attributes through access methods. In this section, we discuss another technique—defining special methods that customize the behavior of direct attribute access.

Python provides three special methods (Fig. 8.2) that a class can define to control how the dot access operator behaves on objects of the class. This technique of redefining an operator's behavior is called "operator overloading," a topic we discuss in detail in the next several sections. Overloading the dot access operator combines the two attribute access techniques we discussed in the previous chapter—a client may access the attributes directly (i.e., through the dot access operator), but doing so actually executes code that performs the operations of access methods.

Figure 8.3 contains a modified definition of class **`Time`**, the class we used to explore attribute access in the previous chapter. The new definition uses special methods **`__getattr__`** and **`__setattr__`** to control how a client accesses and modifies an object's attributes.

Lines 7–13 contain a default constructor for class **`Time`**. The constructor simply assigns the argument values to the new object's attributes. If a class defines special method **`__setattr__`**, Python calls this method every time a program makes an assignment to an object's attribute through the dot operator. Therefore, the statement in line 11 actually results in the call

```
self.__setattr__( "hour", hour )
```

Method	Description
`__delattr__`	Executes when a client deletes an attribute (e.g., **`del anObject.attribute`**)
`__getattr__`	Executes when a client accesses an attribute name that cannot be located in the object's **`__dict__`** attribute (e.g., **`anObject.unfoundName`**)
`__setattr__`	Executes when a client assigns a value to an object's attribute (e.g., **`anObject.attribute = value`**)

Fig. 8.2 Attribute access customization methods.

```python
# Fig: 8.3: TimeAccess.py
# Class Time with customized attribute access.

class Time:
   """Class Time with customized attribute access"""

   def __init__( self, hour = 0, minute = 0, second = 0 ):
      """Time constructor initializes each data member to zero"""

      # each statement invokes __setattr__
      self.hour = hour
      self.minute = minute
      self.second = second

   def __setattr__( self, name, value ):
      """Assigns a value to an attribute"""

      if name == "hour":

         if 0 <= value < 24:
            self.__dict__[ "_hour" ] = value
         else:
            raise ValueError, "Invalid hour value: %d" % value

      elif name == "minute" or name == "second":

         if 0 <= value < 60:
            self.__dict__[ "_" + name ] = value
         else:
            raise ValueError, "Invalid %s value: %d" % \
               ( name, value )

      else:
         self.__dict__[ name ] = value

   def __getattr__( self, name ):
      """Performs lookup for unrecognized attribute name"""

      if name == "hour":
         return self._hour
      elif name == "minute":
         return self._minute
      elif name == "second":
         return self._second
      else:
         raise AttributeError, name

   def __str__( self ):
      """Returns Time object string in military format"""

      # attribute access does not call __getattr__
      return "%.2d:%.2d:%.2d" % \
         ( self._hour, self._minute, self._second )
```

Fig. 8.3 Customized attribute access—class **Time**. (Part 1 of 2.)

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from TimeAccess import Time
>>> time1 = Time( 4, 27, 19 )
>>> print time1
04:27:19
>>> print time1.hour, time1.minute, time1.second
4 27 19
>>> time1.hour = 16
>>> print time1
16:27:19
>>> time1.second = 90
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeAccess.py", line 30, in __setattr__
    raise ValueError, "Invalid %s value: %d" % \
ValueError: Invalid second value: 90
```

Fig. 8.3 Customized attribute access—class **Time**. (Part 2 of 2.)

Method **__setattr__** (lines 15–34) contains the error-checking code needed to maintain the object's data in a consistent state. The method accepts three arguments—the object reference (**self**), the name of the attribute to set and the value to be assigned to the attribute. Line 18 tests whether the attribute to be set is named **"hour"**. If so, lines 20–23 determine whether the specified value falls within the appropriate range. If the value is in the appropriate range, line 21 assigns the value to attribute **_hour** by accessing the appropriate key-value pair in the object's **__dict__** attribute; otherwise, lines 22–23 raise an exception to indicate an invalid value.

It is important that method **__setattr__** uses an object's **__dict__** attribute to set an object's attributes. If line 21 contained the statement

```
self._hour = value
```

method **__setattr__** would execute again, with the arguments **"_hour"** and **value**, resulting in infinite recursion. Assigning a value through the object's **__dict__** attribute, however, does not invoke method **__setattr__**, but simply inserts the appropriate key–value pair in the object's **__dict__**.

Common Programming Error 8.2

In method **__setattr__***, assigning a value to an object's attribute through the dot access operator results in infinite recursion. Use the object's* **__dict__** *instead.*

Lines 25–31 of method **__setattr__** perform similar tests for when the client attempts to assign a value to attributes **minute** or **second**. If the specified value falls within the appropriate range, the method assigns the value to the object's attribute (either **_minute** or **_second**). If the client attempts to assign a value to an attribute other than **hour**, **minute** or **second**, line 33 assigns the value to the specified attribute name, to preserve Python's default behavior for adding attributes to an object.

Common Programming Error 8.3

Assigning a value to an object's attribute, but mistakenly typing the wrong name for that attribute is a logic error. Python adds a new attribute to the object's namespace with the incorrect name.

Lines 36–46 contain the definition for method `__getattr__`. When a client program contains the expression

```
time1.attribute
```

as an *rvalue* (i.e., the right-hand value in an operator expression), Python first looks in `time1`'s `__dict__` attribute for the attribute name. If the attribute name is in `__dict__`, Python simply returns the attribute's value. If the attribute name is not in the object's `__dict__`, Python generates the call

```
time1.__getattr__( attribute )
```

where ***`attribute`*** is the name of the attribute that the client is attempting to access. The method tests for whether the client is attempting to access `hour`, `minute` or `second` and, if so, returns the value of the appropriate attribute. Otherwise the method raises an exception (line 46).

Software Engineering Observation 8.1

The `__getattr__` definition for every class should raise the `AttributeError` exception if the attribute name cannot be found, to preserve Python's default behavior for locating nonexistent attributes.

The interactive session that follows the class definition in Fig. 8.3 demonstrates the benefit of defining special methods `__getattr__` and `__setattr__`. The client program can access the attributes of an object of class `Time` in a transparent manner, through the dot access operator. The interface to class `Time` appears identical to the interface we presented in the first definition of the class in Chapter 7, but it has the advantage of maintaining data in a consistent state. In Chapter 9, Inheritance, we discuss a similar technique—called *properties*—that enables class authors to specify a method that executes when a client attempts to access or modify a particular attribute.

Software Engineering Observation 8.2

Designers of large systems that require strict access to data should use `__getattr__` and `__setattr__` to ensure data integrity. Developers of large systems that use Python 2.2 can use properties, a more efficient technique to take advantage of the syntax allowed by `__getattr__` and `__setattr__`.

8.4 Operator Overloading

Operators provide programmers with a concise notation for expressing manipulations of objects of built-in types. Programmers can also use operators with objects of a class. Although Python does not allow new operators to be created, it does allow most existing operators to be overloaded such that, when these operators are used with objects of a programmer-defined type, the operators have meaning appropriate to the new types.

Software Engineering Observation 8.3

Operator overloading contributes to Python's extensibility, one of the language's most appealing qualities.

Good Programming Practice 8.2

Use operator overloading when it makes a program clearer than accomplishing the same operations with explicit method calls.

Good Programming Practice 8.3

Avoid excessive or inconsistent use of operator overloading; overloaded operators can make a program cryptic and difficult to read.

Although operator overloading may sound like an exotic capability, most programmers implicitly use overloaded operators regularly. For example, the addition operator (`+`) operates quite differently on integers, floating-point numbers and strings. But addition nevertheless works fine with variables of these types and other built-in types, because the addition operator (`+`) has been overloaded in the Python language itself.

Operators are overloaded by writing a method definition as you normally would, except that the method name corresponds to the Python special method for that operator. For example, the method name `__add__` overloads the addition operator (`+`). To use an operator on an object of a class, the class *must* overload (i.e., define a special method for) that operator.

Overloading is most appropriate for mathematical classes. These often require that a substantial set of operators be overloaded to ensure consistency with the way these mathematical classes are handled in the real world. For example, it would be unusual to overload, for rational numbers, only addition, because other arithmetic operators also are used commonly with rational numbers.

Python is an operator-rich language. Python programmers who understand the meaning and context of each operator are likely to make reasonable choices when it comes to overloading operators for new classes.

Operator overloading provides the same concise expressions for user-defined classes that Python provides with its rich collection of operators for built-in types. However, operator overloading is not automatic; the programmer must write operator overloading methods to perform the desired operations.

Extreme misuses of overloading are possible, such as overloading operator `+` to perform subtraction-like operations or overloading operator `-` to perform multiplication-like operations. Such non-intuitive uses of overloading make a program extremely difficult to comprehend and should be avoided.

Good Programming Practice 8.4

Overload operators to perform the same function or similar functions on objects as the operators perform on objects of built-in types. Avoid nonintuitive uses of operators.

8.5 Restrictions on Operator Overloading

Most Python operators and augmented assignment symbols can be overloaded.[1] These are shown in Fig. 8.4.

1. Two operators cannot be overloaded: `{}` and `lambda`. [*Note:* `lambda` is a keyword that supports functional programming—a technique that is beyond the scope of this book.]

<table>
<tr><th colspan="8">Common operators and augmented assignment statements that can be overloaded</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>**</td><td>/</td><td>//</td><td>%</td><td><<</td></tr>
<tr><td>>></td><td>&</td><td>|</td><td>^</td><td>~</td><td><</td><td>></td><td><=</td></tr>
<tr><td>>=</td><td>==</td><td>!=</td><td>+=</td><td>-=</td><td>*=</td><td>**=</td><td>/=</td></tr>
<tr><td>//=</td><td>%=</td><td><<=</td><td>>>=</td><td>&=</td><td>^=</td><td>|=</td><td>[]</td></tr>
<tr><td>()</td><td>.</td><td>``</td><td>in</td><td></td><td></td><td></td><td></td></tr>
</table>

Fig. 8.4 Operators and augmented assignment statements that can be overloaded.

The precedence of an operator cannot be changed by overloading. However, parentheses can be used to force the order of evaluation of overloaded operators in an expression. The associativity of an operator cannot be changed by overloading.

It is not possible to change the "arity" of an operator (i.e., the number of operands an operator takes)—overloaded unary operators remain unary operators, and overloaded binary operators remain binary operators. Operators **+** and **-** each have both unary and binary versions; these unary and binary versions can be overloaded separately, using different method names. It is not possible to create new operators; only existing operators can be overloaded.

Common Programming Error 8.4

Attempting to change the "arity" of an operator via operator overloading causes a fatal runtime error when the overloaded operator's method executes.

The meaning of how an operator works on objects of built-in types cannot be changed by operator overloading. The programmer cannot, for example, change the meaning of how **+** adds two integers. Operator overloading works only with objects of user-defined classes or with a mixture of an object of a user-defined class and an object of a built-in type.

Overloading a binary mathematical operator (e.g., **+**, **-**, *****) automatically overloads the operator's corresponding augmented assignment statement. For example, overloading an addition operator to allow statements like

```
object2 = object2 + object1
```

implies that the **+=** augmented assignment statement also is overloaded to allow statements such as

```
object2 += object1
```

Although (in this case) the programmer does not have to define a method to overload the **+=** assignment statement, such behavior also can be achieved by defining the method explicitly for that class.

Performance Tip 8.1

Sometimes it is preferable to overload an augmented assignment version of an operator to perform the operation "in place" (i.e., without using extra memory by creating a new object).

8.6 Overloading Unary Operators

A unary operator for a class is overloaded as a method that takes only the object reference argument (**self**). When overloading a unary operator (such as **~**) as a method, if **object1** is an object of class **Class**, when the interpreter encounters the expression

```
~object1
```

the interpreter generates the call

```
object1.__invert__()
```

The operand **object1** is the object for which the **Class** method **__invert__** is invoked. Figure 8.5 lists the unary operators and their corresponding special methods.

8.7 Overloading Binary Operators

A binary operator or statement for a class is overloaded as a method with two arguments: **self** and **other**. Later in this chapter, we will overload the **+** operator to indicate addition of two objects of class **Rational**. When overloading binary operator **+**, if **y** and **z** are objects of class **Rational**, then **y + z** is treated as if **y.__add__(z)** had been written, invoking the **__add__** method. If **y** is not an object of class **Rational**, but **z** is an object of class **Rational**, then **y + z** is treated as if **z.__radd__(y)** had been written. The method is named **__radd__**, because the object for which the method executes appears to the right of the operator. Usually, overloaded binary operator methods create and return new objects of their corresponding class.

When overloading assignment statement **+=** as a **Rational** method that accepts two arguments, if **y** and **z** are objects of class **Rational**, then **y += z** is treated as if **y.__iadd__(z)** had been written, invoking the **__iadd__** method. The method is named **__iadd__**, because the method performs its operations "in-place" (i.e., the method uses no extra memory to perform its behavior). Usually, this means that the method performs any necessary calculations on the object reference argument (**self**), then returns the updated reference. Figure 8.6 lists the binary operators and assignment statements and their corresponding special methods.

What happens if we evaluate the expression **y + z** or the statement **y += z**, and only **y** is an object of class **Rational**? In both cases, **z** must be *coerced* (i.e., converted) to an object of class **Rational**, before the appropriate operator overloading method executes. We cover coercion and the special methods that provide coercion behavior in Section 8.9.

Unary operator	Special method
-	**__neg__**
+	**__pos__**
~	**__invert__**

Fig. 8.5 Unary operators and their corresponding special methods.

Binary operator/ statement	Special method
+	__**add**__, __**radd**__
-	__**sub**__, __**rsub**__
*****	__**mul**__, __**rmul**__
/	__**div**__, __**rdiv**__, __**truediv**__ (for Python 2.2), __**rtruediv**__ (for Python 2.2)
//	__**floordiv**__, __**rfloordiv**__ (for Python version 2.2)
%	__**mod**__, __**rmod**__
******	__**pow**__, __**rpow**__
<<	__**lshift**__, __**rlshift**__
>>	__**rshift**__, __**rrshift**__
&	__**and**__, __**rand**__
^	__**xor**__, __**rxor**__
|	__**or**__, __**ror**__
+=	__**iadd**__
-=	__**isub**__
***=**	__**imul**__
/=	__**idiv**__, __**itruediv**__ (for Python version 2.2)
//=	__**ifloordiv**__ (for Python version 2.2)
%=	__**imod**__
****=**	__**ipow**__
<<=	__**ilshift**__
>>=	__**irshift**__
&=	__**iand**__
^=	__**ixor**__
|=	__**ior**__
==	__**eq**__
!+, <>	__**ne**__
>	__**gt**__
<	__**lt**__
>=	__**ge**__
<=	__**le**__

Fig. 8.6 Binary operators and their corresponding special methods.

8.8 Overloading Built-in Functions

A class also may define special methods that execute when certain built-in functions are called on an object of the class. For example, we may define special method __***abs***__ for

class **Rational**, to execute when a program calls **abs(rationalObject)** to compute the absolute value of an object of that class. The table in Fig. 8.7 contains a list of common built-in functions and the corresponding special methods that the class may define.

8.9 Converting Between Types

Most programs process information of a variety of types. Sometimes all the operations "stay within a type." For example, adding (concatenating) a string to a string produces a string. But, it is often necessary to convert or *coerce* data of one type to data of another type. This can happen in assignments and in calculations. The interpreter knows how to perform certain conversions among built-in types. Programmers can force conversions among built-in types by calling the appropriate Python function, such as **int** or **float**.

But what about user-defined classes? The interpreter cannot know how to convert among user-defined classes and built-in types. The programmer must specify how such conversions are to occur with special methods that override the appropriate Python functions. For example, a class can define special method **__*int*__** that overloads the behavior of the call **int(*anObject*)** to return an integer representation of the object. The table in Fig. 8.8 lists the special methods that a class may define to implement type coercion. Each special method has a corresponding built-in function.

Built-in Function	Description	Special method
abs(*x* **)**	Returns the absolute value of *x*.	**__abs__**
divmod(*x*, *y* **)**	Returns a tuple that contains the integer and remainder components of *x* **%** *y*.	**__divmod__**
len(*x* **)**	Returns the length of *x* (*x* should be a sequence).	**__len__**
pow(*x*, *y*[, *z*] **)**	Returns the result of x^y. With three arguments, returns (x^y) **%** *z*.	**__pow__**
repr(*x* **)**	Returns a formal string representation of x (i.e., a string from which object *x* can be replicated).	**__repr__**

Fig. 8.7 Common built-in functions and their corresponding special methods.

Method	Description
__coerce__	Converts two values to the same type.
__complex__	Converts object to complex number type.
__float__	Converts object to floating-point number type.
__hex__	Converts object to hexidecimal string type.

Fig. 8.8 Coercion methods. (Part 1 of 2.)

Method	Description
`__int__`	Converts object to integer number type.
`__long__`	Converts object to long integer number type.
`__oct__`	Converts object to octal string type.
`__str__`	Converts object to string type. Also used to obtain informal string representation of object (i.e., a string that simply describes object).

Fig. 8.8 Coercion methods. (Part 2 of 2.)

8.10 Case Study: A `Rational` Class

Figure 8.9 illustrates a **`Rational`** class. The class uses overloaded numerical operators, built-in functions and statements to manipulate rational numbers. A rational number is a fraction represented as a numerator (top) and a denominator (bottom). A rational number can be positive, negative or zero. Class **`Rational`**'s interface includes a default constructor, string representation method, overloaded **`abs`** function, equality operators and several mathematical operators. The class also defines one method **`simplify`** that reduces the rational number. Reducing a rational number is the process of dividing the numerator and denominator by their greatest common divisor, to express the rational number in "simplest terms." The file defines a **`gcd`** function, used by class **`Rational`** to compute the greatest common divisor of two values.

In the class definition (Fig. 8.9), lines 4–12 define function **`gcd`**, which computes the greatest common divisor of two values. Class **`Rational`** uses this function to simplify the rational number.

The **`Rational`** constructor (lines 17–30) takes two arguments—**`top`** and **`bottom`**—that default to 1. If the client attempts to create an object of class **`Rational`** with denominator 0, the constructor raises an exception (***`ZeroDivisionError`***) to indicate an error (lines 21–22). **`ZeroDivisionError`** is the name of an exception object that Python places in the built-in namespace when the interpreter begins. We discuss this exception and others (e.g., **`IndexErrror`**, **`KeyError`**, etc.) in Chapter 12, Exception Handling. Lines 25–26 assign the object's numerator and denominator as the absolute value of the arguments passed to the constructor. Lines 27–28 compute and assign the object's sign to attribute **`sign`**. Line 30 calls method **`simplify`**, to reduce the rational number to its simplest form.

```
1  # Fig. 8.9: RationalNumber.py
2  # Definition of class Rational.
3
4  def gcd( x, y ):
5     """Computes greatest common divisor of two values"""
6
7     while y:
8        z = x
9        x = y
10       y = z % y
```

Fig. 8.9 Operator overloading—**`Rational.py`**. (Part 1 of 4.)

```python

   return x

class Rational:
   """Representation of rational number"""

   def __init__( self, top = 1, bottom = 1 ):
      """Initializes Rational instance"""

      # do not allow 0 denominator
      if bottom == 0:
         raise ZeroDivisionError, "Cannot have 0 denominator"

      # assign attribute values
      self.numerator = abs( top )
      self.denominator = abs( bottom )
      self.sign = ( top * bottom ) / ( self.numerator *
         self.denominator )

      self.simplify()  # Rational represented in reduced form

   # class interface method
   def simplify( self ):
      """Simplifies a Rational number"""

      common = gcd( self.numerator, self.denominator )
      self.numerator /= common
      self.denominator /= common

   # overloaded unary operator
   def __neg__( self ):
      """Overloaded negation operator"""

      return Rational( -self.sign * self.numerator,
                       self.denominator )

   # overloaded binary arithmetic operators
   def __add__( self, other ):
      """Overloaded addition operator"""

      return Rational(
         self.sign * self.numerator * other.denominator +
         other.sign * other.numerator * self.denominator,
         self.denominator * other.denominator )

   def __sub__( self, other ):
      """Overloaded subtraction operator"""

      return self + ( -other )

   def __mul__( self, other ):
      """Overloaded multiplication operator"""

```

Fig. 8.9 Operator overloading—**Rational.py**. (Part 2 of 4.)

```
      return Rational( self.numerator * other.numerator,
                       self.sign * self.denominator *
                       other.sign * other.denominator )

   def __div__( self, other ):
      """Overloaded / division operator."""

      return Rational( self.numerator * other.denominator,
                       self.sign * self.denominator *
                       other.sign * other.numerator )

   def __truediv__( self, other ):
      """Overloaded / division operator. (For use with Python
      versions (>= 2.2) that contain the // operator)"""

      return self.__div__( other )

   # overloaded binary comparison operators
   def __eq__( self, other ):
      """Overloaded equality operator"""

      return ( self - other ).numerator == 0

   def __lt__( self, other ):
      """Overloaded less-than operator"""

      return ( self - other ).sign < 0

   def __gt__( self, other ):
      """Overloaded greater-than operator"""

      return ( self - other ).sign > 0

   def __le__( self, other ):
      """Overloaded less-than or equal-to operator"""

      return ( self < other ) or ( self == other )

   def __ge__( self, other ):
      """Overloaded greater-than or equal-to operator"""

      return ( self > other ) or ( self == other )

   def __ne__( self, other ):
      """Overloaded inequality operator"""

      return not ( self == other )

   # overloaded built-in functions
   def __abs__( self ):
      """Overloaded built-in function abs"""

      return Rational( self.numerator, self.denominator )
```

Fig. 8.9 Operator overloading—**Rational.py**. (Part 3 of 4.)

```python

   def __str__( self ):
      """String representation"""

      # determine sign display
      if self.sign == -1:
         signString = "-"
      else:
         signString = ""

      if self.numerator == 0:
         return "0"
      elif self.denominator == 1:
         return "%s%d" % ( signString, self.numerator )
      else:
         return "%s%d/%d" % \
            ( signString, self.numerator, self.denominator )

   # overloaded coercion capability
   def __int__( self ):
      """Overloaded integer representation"""

      return self.sign * divmod( self.numerator,
         self.denominator )[ 0 ]

   def __float__( self ):
      """Overloaded floating-point representation"""

      return self.sign * float( self.numerator ) / self.denominator

   def __coerce__( self, other ):
      """Overloaded coercion. Can only coerce int to Rational"""

      if type( other ) == type( 1 ):
         return ( self, Rational( other ) )
      else:
         return None
```

Fig. 8.9 Operator overloading—**Rational.py**. (Part 4 of 4.)

```python
# Fig. 8.10: fig08_10.py
# Driver for class Rational.

from RationalNumber import Rational

# create objects of class Rational
rational1 = Rational()  # 1/1
rational2 = Rational( 10, 30 )  # 10/30 (reduces to 1/3)
rational3 = Rational( -7, 14 )  # -7/14 (reduces to -1/2)

# print objects of class Rational
print "rational1:", rational1
```

Fig. 8.10 Operator overloading—**fig08_10.py**. (Part 1 of 2.)

```
print "rational2:", rational2
print "rational3:", rational3
print

# test mathematical operators
print rational1, "/", rational2, "=", rational1 / rational2
print rational3, "-", rational2, "=", rational3 - rational2
print rational2, "*", rational3, "-", rational1, "=", \
   rational2 * rational3 - rational1

# overloading + implicitly overloads +=
rational1 += rational2 * rational3
print "\nrational1 after adding rational2 * rational3:", rational1
print

# test comparison operators
print rational1, "<=", rational2, ":", rational1 <= rational2
print rational1, ">", rational3, ":", rational1 > rational3
print

# test built-in function abs
print "The absolute value of", rational3, "is:", abs( rational3 )
print

# test coercion
print rational2, "as an integer is:", int( rational2 )
print rational2, "as a float is:", float( rational2 )
print rational2, "+ 1 =", rational2 + 1
```

```
rational1: 1
rational2: 1/3
rational3: -1/2

1 / 1/3 = 3
-1/2 - 1/3 = -5/6
1/3 * -1/2 - 1 = -7/6

rational1 after adding rational2 * rational3: 5/6

5/6 <= 1/3 : 0
5/6 > -1/2 : 1

The absolute value of -1/2 is: 1/2

1/3 as an integer is: 0
1/3 as a float is: 0.333333333333
1/3 + 1 = 4/3
```

Fig. 8.10 Operator overloading—**fig08_10.py**. (Part 2 of 2.)

Method **simplify** (lines 33–38) reduces an object of class **Rational**. The method first calls function **gcd** to determine the greatest common divisor of the object's numerator and denominator (line 36). The method then uses the greatest common divisor to simplify the rational object (lines 37–38).

Method **__neg__** (lines 41–45) overloads the unary negation operator. If **rational** is an object of class **Rational**, when the interpreter encounters the expression

```
-rational
```

the interpreter generates method call

```
rational.__neg__()
```

which simply creates a new object of class **Rational** with the negated sign of the original object.

Method **__add__** (lines 48–54) overloads the addition operator. This method takes two arguments—the object reference (**self**), and a reference to another object of class **Rational**. If **rational1** and **rational2** are two objects of class **Rational**, when the interpreter encounters the expression

```
rational1 + rational2
```

the interpreter generates method call

```
rational1.__add__( rational2 )
```

This method creates and returns a new object of class **Rational** that represents the results of adding **self** to **other**. The numerator of this new value is computed with the expression

```
self.sign * self.numerator * other.denominator +
other.sign * other.numerator * self.denominator
```

and the denominator is computed with the expression

```
self.denominator * other.denominator
```

Method **__sub__** (lines 56–59) overloads the binary subtraction operator. This method uses the overloaded **+** and **-** operators to create and return the results of subtracting the method's second argument from the method's first argument.

Method **__mul__** (lines 61–66) overloads the binary multiplication operator. This method creates and returns a new object of class **Rational** that represents the product of the method's two arguments.

Method **__div__** (lines 68–73) overloads the binary division operator **/** and creates and returns a new object of class **Rational** that represents the results of dividing the method's two arguments. Method **__truediv__** (lines 75–79) overloads the binary division operator **/** for Python versions 2.2 and greater that use floating-point division. This method simply calls method **__div__**, because the **/** operator should perform the same operation, regardless of the Python version. [*Note:* See Chapter 2, Introduction to Python Programming, for more information on the difference in the **/** operator between Python versions.]

Method **__eq__** (lines 82–85) overloads the binary equality operator (**==**). If **rational1** and **rational2** are two objects of class **Rational**, when the interpreter encounters the expression

```
rational1 == rational2
```

the interpreter generates method call

```
rational1.__eq__( rational2 )
```

This method subtracts the two objects and determines whether the numerator of the result is 0. **Rational** objects are reduced to their simplest form when created; therefore, we do not need to reduce the method's argument values before testing whether they are equal.

Method **__lt__** (lines 87–90) overloads the binary less-than operator (**<**). This method subtracts its second argument from its first argument and tests whether the sign of the result is less than 0. Method **__gt__** (lines 92–95) overloads the binary greater-than operator (**>**). This method subtracts its second argument from its first and tests whether the sign of the result is greater than 0.

Methods **__le__** (lines 97–100), **__ge__** (lines 102–105) and **__ne__** (lines 107–110) overload the **<=**, **>=** and inequality operators (**!=** and **<>**) for objects of class **Rational**. These methods use the overloaded equality (**==**), greater-than (**>**) and less-than (**<**) operators to performs their operations.

Lines 113–116 define special method **__abs__** to overload the functionality of the built-in **abs** function. If **rational** is an object of class **Rational**, when the interpreter encounters the expression

```
abs( rational )
```

the interpreter generates the method call

```
rational.__abs__()
```

This method creates a new object of class **Rational** using the values of the numerator and denominator of the object reference argument (recall that the constructor stores these values as positive integers).

Lines 118–133 define method **__str__** so that clients may use the **print** statement or built-in function **str** to display information about an object of class **Rational**. If the object's numerator is 0, **__str__** returns the string representation of integer value 0; if the object's denominator is 1, **__str__** returns the string representation of the object's sign and numerator. Otherwise, the method returns the string representation of the object's sign, followed by the string representation of the object's numerator, followed by **"/"**, followed by the string representation of the object's denominator.

Lines 136–153 define special methods for coercion behavior. Method **__int__** (lines 136–140) executes when a client invokes built-in function **int** on an object of class **Rational**. The method calls built-in function **divmod** to compute the integer division and remainder components of dividing the numerator by the denominator. The method returns the first element in the tuple returned from **divmod**, which represents the integer division component. Method **__float__** (lines 142–145) executes when a client invokes built-in function **float** on an object of class **Rational**. The method multiplies the object's sign (-1 or 1) by the result of dividing the numerator by the denominator and ensures a floating-point return value by call function **float** on the numerator.

Method **__coerce__** (lines 147–153) executes when a client calls built-in function **coerce** on an object of class **Rational** and another object or when the client performs so-called "mixed-mode" arithmetic. An example of mixed-mode arithmetic is the statement

```
rational + 1
```

which attempts to add an integer to an object of class **Rational**. This statement results in the method call

```
rational.__add__( rational.__coerce__( 1 ) )
```

Special method **__coerce__** should contain code that converts the object and the other type to the same type and should return a tuple that contains the two converted values. Method **__coerce__** for class **Rational** converts only integer values. Line 150 determines whether the type of the method's second argument is an integer. If so, the method returns a tuple that contains the object reference argument and a new object of class **Rational**, created by passing the integer argument to **Rational**'s constructor. Python expects special method **__coerce__** to return **None** if a coercion of the two types is not possible; therefore, line 153 returns **None** if the method's argument is not an integer.

The driver program (Fig. 8.10) creates objects of class **Rational**—**rational1** is initialized by default to 1/1, **rational2** is initialized to 10/30 and **rational3**, which is initialized to -7/14. The **Rational** constructor calls method **simplify** to reduce the specified numerator and denominator. Thus, **rational2** represents the value 1/3, and **rational3** represents the value -1/2.

The driver program outputs each of the constructed objects of class **Rational**, using the **print** statement. Lines 17–21 demonstrate the results of using overloaded arithmetic operators **/**, **-** and *****. Lines 24–26 demonstrate that overloading the **+** addition operator implicitly overloads the **+=** assignment statement. The program uses the **+=** augmented assignment statement to add to **rational1** the product of **rational2 * rational3**, then prints the results. The driver then prints the results of comparing the objects of class **Rational** through the overloaded comparison operators (lines 29–31). Line 34 prints the absolute value of object **rational3**. Lines 38-40 tests **Rational**'s coercion capability by printing the integer representation (invoking method **__int__**) and the floating-point representation (invoking method **__float__**) and by adding an object of class **Rational** and an integer (invoking method **__coerce__**).

8.11 Overloading Sequence Operations

We have seen how to use special methods to define a class that behaves like a numeric type. A class also can define several special methods to implement sequence operations, providing a list-based interface to its clients. An object of the class can provide access to its elements through subscripts and slices, can be passed to function **len** to determine its length and can support the operators and provide the methods that lists support. The table in Fig. 8.11 contains some methods that a sequence class should provide. In the next section, we define several of these methods for a list-based class that contains only unique values.

Method	Description
__add__, **__radd__**, **__iadd__**	Overloads addition operator for concatenating sequences (e.g., **sequence1 + sequence2**)

Fig. 8.11 Sequence methods. (Part 1 of 2.)

Method	Description
`append`	Called to append an element to a mutable sequence (e.g., `sequence.append( element )`)
`__contains__`	Called to test for membership (e.g., `element in sequence`)
`count`	Called to determine number of occurrences of element in a mutable sequence (e.g., `sequence.count( element )`)
`__delitem__`	Called to delete an item from a mutable sequence (e.g., `del sequence[ index ]`)
`__getitem__`	Called for subscript access (e.g., `sequence[ index ]`)
`index`	Called to obtain index of first occurrence of an element in a mutable sequence (e.g., `sequence.index( element )`)
`insert`	Called to insert an element at a given index in a mutable sequence (e.g., `sequence.insert( index, element )`)
`__len__`	Called for length of sequence (e.g., `len( sequence )`)
`__mul__`, `__rmul__`, `__imul__`	Overloads multiplication operator for repeating sequences (e.g., `sequence * 3`)
`pop`	Called to remove an element from a mutable sequence (e.g., `sequence.pop()`)
`remove`	Called to remove first occurrence of a value from a mutable sequence (e.g., `sequence.remove()`)
`reverse`	Called to reverse a mutable sequence in place (e.g., `sequence.reverse()`)
`__setitem__`	Called for assignment to a mutable sequence (e.g., `sequence[ index ] = value`)
`sort`	Called to sort a mutable sequence in place (e.g., `sequence.sort()`)

Fig. 8.11 Sequence methods. (Part 2 of 2.)

8.12 Case Study: A `SingleList` Class

We now present an example of a class that "wraps" (contains) a list to illustrate how to define several special methods to create a class that behaves like a sequence. The list allows clients to insert only new (unique) values in the list and allows the list to be displayed in tabular form. This example will sharpen your appreciation of data abstraction. You will probably want to suggest enhancements to this example. Class development is an interesting, creative and intellectually challenging activity—always with the goal of "crafting valuable classes."

The program of Fig. 8.12 demonstrates class `SingleList` and its overloaded operators, statements and other special methods. First we walk through the driver program (Fig. 8.13). Then we consider the class definition and each of the class's methods.

```python
# Fig. 8.12: NewList.py
# Simple class SingleList.

class SingleList:

   def __init__( self, initialList = None ):
      """Initializes SingleList instance"""

      self.__list = []  # internal list, contains no duplicates

      # process list passed to __init__, if necessary
      if initialList:

         for value in initialList:

            if value not in self.__list:
               self.__list.append( value )  # add original value

   # string representation method
   def __str__( self ):
      """Overloaded string representation"""

      tempString = ""
      i = 0

      # build output string
      for i in range( len( self ) ):
         tempString += "%12d" % self.__list[ i ]

         if ( i + 1 ) % 4  == 0:  # 4 numbers per row of output
            tempString += "\n"

      if i % 4 != 0:  # add newline, if necessary
         tempString += "\n"

      return tempString

   # overloaded sequence methods
   def __len__( self ):
      """Overloaded length of the list"""

      return len( self.__list )

   def __getitem__( self, index ):
      """Overloaded sequence element access"""

      return self.__list[ index ]

   def __setitem__( self, index, value ):
      """Overloaded sequence element assignment"""

```

Fig. 8.12 `SingleList` class with operator overloading—`SingleList.py`. (Part 1 of 2.)

```
         if value in self.__list:
            raise ValueError, \
               "List already contains value %s" % str( value )

         self.__list[ index ] = value

   # overloaded equality operators
   def __eq__( self, other ):
      """Overloaded == operator"""

      if len( self ) != len( other ):
         return 0  # lists of different sizes

      for i in range( 0, len( self ) ):

         if self.__list[ i ] != other.__list[ i ]:
            return 0  # lists are not equal

      return 1  # lists are equal

   def __ne__( self, other ):
      """Overloaded != and <> operators"""

      return not ( self == other )
```

Fig. 8.12 `SingleList` class with operator overloading—`SingleList.py`. (Part 2 of 2.)

```
# Fig. 8.13: fig08_13.py
# Driver for simple class SingleList.

from NewList import SingleList

def getIntegers():
   size = int( raw_input( "List size: " ) )

   returnList = []  # the list to return

   for i in range( size ):
      returnList.append(
         int( raw_input( "Integer %d: " % ( i + 1 ) ) ) )

   return returnList

# input and create integers1 and integers2
print "Creating integers1..."
integers1 = SingleList( getIntegers() )

print "Creating integers2..."
integers2 = SingleList( getIntegers() )
```

Fig. 8.13 `SingleList` class with operator overloading—`fig08_13.py`. (Part 1 of 3.)

```
# print integers1 size and contents
print "\nSize of list integers1 is", len( integers1 )
print "List:\n", integers1

# print integers2 size and contents
print "\nSize of list integers2 is", len( integers2 )
print "List:\n", integers2

# use overloaded comparison operator
print "Evaluating: integers1 != integers2"

if integers1 != integers2:
   print "They are not equal"

print "\nEvaluating: integers1 == integers2"

if integers1 == integers2:
   print "They are equal"

print "integers1[ 0 ] is", integers1[ 0 ]
print "Assigning 0 to integers1[ 0 ]"
integers1[ 0 ] = 0
print "integers1:\n", integers1
```

```
Creating integers1...
List size: 8
Integer 1: 1
Integer 2: 2
Integer 3: 3
Integer 4: 4
Integer 5: 5
Integer 6: 6
Integer 7: 7
Integer 8: 8
Creating integers2...
List size: 10
Integer 1: 9
Integer 2: 10
Integer 3: 11
Integer 4: 12
Integer 5: 13
Integer 6: 14
Integer 7: 15
Integer 8: 16
Integer 9: 17
Integer 10: 18

Size of list integers1 is 8
List:
           1           2           3           4
           5           6           7           8
```

Fig. 8.13 `SingleList` class with operator overloading—`fig08_13.py`. (Part 2 of 3.)

```
Size of list integers2 is 10
List:
           9          10          11          12
          13          14          15          16
          17          18

Evaluating: integers1 != integers2
They are not equal

Evaluating: integers1 == integers2
integers1[ 0 ] is 1
Assigning 0 to integers1[ 0 ]
integers1:
           0           2           3           4
           5           6           7           8
```

Fig. 8.13 `SingleList` class with operator overloading—`fig08_13.py`. (Part 3 of 3.)

The program (Fig. 8.13) begins by creating two objects of class **`SingleList`** (lines 18–22). This class's constructor takes a list as an argument. To create this list, we call function **`getIntegers`** (lines 6–15). This function prompts the user to enter integers and returns a list of these integers. Lines 25–26 use overloaded Python function **`len`** to determine the size of **`integers1`** and use the **`print`** statement (which implicitly calls method **`__str__`**) to confirm that the list elements were initialized correctly by the constructor. Next, lines 29–30 output the size and contents of **`integers2`**.

Lines 35–41 test the overloaded equality operator (**`==`**) and inequality operator (**`!=`**) by first evaluating the condition

```
integers1 != integers2
```

The program prints a message if the two objects are not equal (line 36). Similarly, line 41 prints a message if the two objects are identical.

Line 43 uses the overloaded subscript operator to refer to **`integers1[ 0 ]`**. This subscripted name is used as an *rvalue* to print the value in **`integers1[ 0 ]`**. Line 45 uses **`integers1[ 0 ]`** as an *lvalue* on the left side of an assignment statement to assign a new value, **`0`**, to element 0 of **`integers1`**.

Now that we have seen how this program operates, let us walk through the class's method definitions (Fig. 8.12). Lines 6–17 define the constructor for the class. The constructor initializes attribute **`_list`** to be the empty list. If the user specified a value for parameter **`initialList`**, the constructor inserts all unique elements from **`initialList`** into **`_list`**.

Lines 20–36 define method **`__str__`** for representing objects of class **`IntegerList`** as a string. This method builds a string (**`tempString`**) by iterating over the elements in the list and formatting the elements in tabular format, with four elements in each row. Line 36 returns the formatted string.

Lines 39–42 define method **`__len__`**, which overrides the Python **`len`** function. When the interpreter encounters the expression

```
len( integers1 )
```

in the driver program, the interpreter generates the call

```
integers1.__len__()
```

This method simply returns the length of attribute **__list**.

Lines 44–56 define two overloaded subscript operators for the class. When the interpreter encounters the expression

```
integers1[ 0 ]
```

in the driver program, the interpreter invokes the appropriate method by generating the call

```
integers1.__getitem__( 0 )
```

to return the value of element 0 (e.g., line 43 in the driver program), or the call

```
integers1.__setitem__( 0, value )
```

to set the value of a list element (e.g., line 45 in the driver program). When the **[]** operator is used in an *rvalue* expression, method **__getitem__** is called; when the **[]** operator is used in an *lvalue* expression, method **__setitem__** is called.

Method **__getitem__** (lines 44–47) simply returns the value of the appropriate element. Method **__setitem__** (lines 49–56) first ascertains whether the list already contains the new element. If the list contains the new element, the method raises an exception; otherwise, the method sets the new value. Because **SingleList** methods manipulate a basic list, any out-of-range errors that apply to regular list data types apply to our **SingleList** type.

Lines 59–70 define the overloaded equality operator (**==**) for the class. When the interpreter encounters the expression

```
integers1 == integers2
```

the interpreter invokes the **__eq__** method by generating the call

```
integers1.__eq__( integers2 )
```

The **__eq__** method immediately returns 0 if the length of the lists are different (lines 62–63). Otherwise, the method compares each pair of elements (lines 65–68). If they are all the same, the method returns 1 (line 70). The first pair of elements to differ causes the method to return 0 immediately (line 68). Line 72–75 define method **__ne__** for testing whether two **NewLists** are unequal. The method simply uses the overloaded **==** operator to determine whether the two objects are unequal.

Class **SingleList** defines only some of the methods suggested for sequences in Fig. 8.11. The exercises contain instructions for implementing some of the remaining methods.

8.13 Overloading Mapping Operations

Python defines several special methods to provide a mapping-based interface to its clients. An object of a class that implements these methods can provide access to its elements through subscripts, can be passed to function **len** to determine the object's length (i.e., the

number of key–value pairs) and can support the methods that dictionaries support. The table in Fig. 8.14 contains some methods that a mapping class should provide. In the next section, we show an example of a class that defines many of these methods, to provide a dictionary interface to a basic object.

8.14 Case Study: A `SimpleDictionary` Class

Recall that an object of a class has a namespace that contains identifiers and their values. Attribute **`__dict__`** contains this information for each object. We can take advantage of this fact to provide a dictionary interface to every object of a class. Figure 8.15 demonstrates class **`SimpleDictionary`**, which defines special methods to implement mapping behaviors of the class.

Method	Description
`clear`	Called to remove all items from mapping (e.g., `mapping.clear()`)
`__contains__`	Called to test for membership; should return same value as method `has_key` (e.g., `key in mapping`) [*Note:* This is new for Python 2.2 dictionaries.]
`copy`	Called to return a shallow copy of mapping (e.g., `mapping.copy()`)
`__delitem__`	Called to delete an item from mapping (e.g., `del mapping[ key ]`)
`get`	Called to obtain the value of a key in mapping (e.g., `mapping.get( key )`)
`__getitem__`	Called for subscript access through key (e.g., `mapping[ key ]`)
`has_key`	Called to determine if mapping contains a key (e.g., `mapping.has_key( key )`)
`items`	Called to obtain a list of key-value pairs in mapping (e.g., `mapping.items()`)
`keys`	Called to obtain a list of keys in mapping (e.g., `mapping.keys()`)
`__len__`	Called for length of mapping (e.g., `len( mapping )`)
`__setitem__`	Called for insertion or assignment through key (e.g., `mapping[ key ] = value`)
`values`	Called to return a list of values in mapping (e.g., mapping.values())
`update`	Called to insert items from another mapping (e.g., `mapping.update( otherMapping )`)

Fig. 8.14 Mapping methods.

```python
# Fig. 8.15: NewDictionary.py
# Definition of class SimpleDictionary.

class SimpleDictionary:
   """Class to make an instance behave like a dictionary"""

   # mapping special methods
   def __getitem__( self, key ):
      """Overloaded key-value access"""

      return self.__dict__[ key ]

   def __setitem__( self, key, value ):
      """Overloaded key-value assignment/creation"""

      self.__dict__[ key ] = value

   def __delitem__( self, key ):
      """Overloaded key-value deletion"""

      del self.__dict__[ key ]

   def __str__( self ):
      """Overloaded string representation"""

      return str( self.__dict__ )

   # common mapping methods
   def keys( self ):
      """Returns list of keys in dictionary"""

      return self.__dict__.keys()

   def values( self ):
      """Returns list of values in dictionary"""

      return self.__dict__.values()

   def items( self ):
      """Returns list of items in dictionary"""

      return self.__dict__.items()
```

Fig. 8.15 Mapping interface—class `SimpleDictionary`.

```python
# Fig. 8.16: fig08_16.py
# Driver for class SimpleDictionary.

from NewDictionary import SimpleDictionary

# create and print SimpleDictionary object
simple = SimpleDictionary()
```

Fig. 8.16 Mapping interface—`fig08_16.py`.

```
8  print "The empty dictionary:", simple
9
10 # add values to simple (invokes simple.__setitem__)
11 simple[ 1 ] = "one"
12 simple[ 2 ] = "two"
13 simple[ 3 ] = "three"
14 print "The dictionary after adding values:", simple
15
16 del simple[ 1 ]  # remove a value
17 print "The dictionary after removing a value:", simple
18
19 # use mapping methods
20 print "Dictionary keys:", simple.keys()
21 print "Dictionary values:", simple.values()
22 print "Dictionary items:", simple.items()
```

```
The empty dictionary: {}
The dictionary after adding values: {1: 'one', 2: 'two', 3: 'three'}
The dictionary after removing a value: {2: 'two', 3: 'three'}
Dictionary keys: [2, 3]
Dictionary values: ['two', 'three']
Dictionary items: [(2, 'two'), (3, 'three')]
```

Fig. 8.16 Mapping interface—**fig08_16.py**.

Each method in the class (Fig. 8.15) simply calls the appropriate method for the object's **__dict__** attribute. Method **__getitem__** (lines 8–11) accepts a key argument that contains the key value to retrieve from the dictionary. Line 11 simply uses the **[]** operator to retrieve the specified key from the object's **__dict__**. Method **__setitem__** (lines 13–16) accepts as arguments a key and a value. The method simply inserts or updates the key-value pair in the object's **__dict__**. Method **__delitem__** (lines 18–21) executes when the client uses keyword **del** to remove a key-value pair from the dictionary. The method simply removes the key-value pair from the object's **__dict__**. Method **__str__** (lines 23–26) returns a string representation of an object of class **SimpleDictionary** by passing the object's **__dict__** to built-in function **str**. Methods **keys** (lines 29–32), **values** (lines 34–37) and **items** (lines 39—42) each return their appropriate value by calling the corresponding method on the object's **__dict__**.

The driver program (Fig. 8.16) creates one object of class **SimpleDictionary** and uses the **print** statement to output the object's value (lines 7–8). Lines 11–13 add new values to the object with the **[]** operator, invoking method **simple.__setitem__**. Line 16 uses keyword **del** to delete an element from the object, invoking method **object.__delitem__**. Lines 20–22 call methods **keys**, **values** and **items**, to print the key-value pairs that the object stores.

In this chapter, we introduced the concept of class customization, wherein a class defines certain special methods to provide a syntax-based interface. These special methods perform a wide variety of tasks in Python, including string representation, attribute access, operator overloading and subscript access. We discussed the methods that provide each of these behaviors, and implemented three case studies that demonstrated how these methods can be used. In the next chapter, we discuss inheritance, a feature that allows programmers

to define new classes that take advantage of the attributes and behaviors of existing classes. This ability is a key advantage of object-oriented programming, because it lets programmers focus only on the new behaviors a class should exhibit. For example, the technique we employed in this chapter of implementing a dictionary interface by calling the methods of an object's underlying `__dict__` attribute leads to some amount of redundant code. With inheritance, we can define a class that "re-uses" the behaviors of the standard dictionary type, without having to define every mapping method explicitly.

SUMMARY

- A special method is a method that has a special meaning in Python; the Python interpreter calls one of an object's special methods when the client performs a certain operation on the object.
- A class author implements special methods to customize the behavior of the class. The purpose of customization is to provide the clients of a class with a simple notation for manipulating objects of the class.
- Operator overloading consists of defining special methods to describe how operators behave with objects of programmer-defined types.
- Python enables programmers to overload most operators to be sensitive to the context in which they are used. The interpreter takes the action appropriate for the manner in which the operator is used.
- A Python class can define special method `__str__`, to provide an informal (i.e., human-readable) string representation of an object of the class. This method executes when a client uses an object with the **`print`** statement, the **`%`** string formatting operator or built-in function **`str`**.
- Python provides three special methods—`__getattr__`, `__setattr__` and `__delattr__`—that a class can define to control how the dot access operator behaves on objects of the class.
- If a class defines special method `__setattr__`, Python calls this method every time a program makes an assignment to an object's attribute through the dot operator.
- Assigning a value through the object's `__dict__` attribute does not invoke method `__setattr__`, but simply inserts the appropriate key–value pair in the object's `__dict__`.
- When a client program accesses an object attribute as an *rvalue*, Python first looks in the object's `__dict__` attribute for the attribute name. If the attribute name is not in `__dict__`, Python invokes the object's `__getattr__` method.
- The `__getattr__` definition for every class should raise the **`AttributeError`** exception if the attribute name cannot be found, to preserve Python's default behavior for looking up nonexistent attributes.
- Although Python does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when these operators are used with objects of a programmer-defined type, the operators have meaning appropriate to the new types.
- Operators are overloaded by writing a method definition as you normally would, except that the method name corresponds to the Python special method for that operator. To use an operator on an object of a class, the class *must* overload (i.e., define a special method for) that operator.
- Operator overloading is not automatic; the programmer must write operator-overloading methods to perform the desired operations.
- The precedence of an operator cannot be changed by overloading.
- It is not possible to change the "arity" of an operator (i.e., the number of operands an operator takes): Overloaded unary operators remain unary operators; overloaded binary operators remain binary operators.

- The meaning of how an operator works on objects of built-in types cannot be changed by operator overloading. Operator overloading works only with objects of user-defined classes or with a mixture of an object of a user-defined class and an object of a built-in type.
- Overloading a binary mathematical operator automatically overloads the operator's corresponding augmented assignment statement, although the programmer can overload the augmented assignment statement explicitly.
- A unary operator for a class is overloaded as a method that takes only the object reference argument (**self**).
- A binary operator or statement for a class is overloaded as a method with two arguments: **self**, and **other**.
- A class also may define special methods that execute when certain built-in functions are called on an object of the class.
- The interpreter knows how to perform certain conversions among built-in types. Programmers can force conversions among built-in types by calling the appropriate function, such as **int** or **float**.
- The programmer must specify how conversions among user-defined classes and built-in types are to occur. Such conversions can be performed with special methods that override the appropriate Python functions.
- Method **__truediv__** overloads the binary division operator **/** for Python versions 2.2 and greater that use floating-point division.
- Method **__coerce__** executes when a client calls built-in function **coerce** on an object of class **Rational** and another object or when the client performs so-called "mixed-mode" arithmetic.
- Special method **__coerce__** should contain code that converts the reference object and the other type to the same type and should return a tuple that contains the two converted values. Python expects special method **__coerce__** to return **None** if a coercion of the two types is not possible.
- A class also can define several special methods to implement sequence operations, providing a list-based interface to its clients.
- When a program accesses an element of a sequence- or dictionary-like object as an *rvalue*, the object's **__getitem__** method executes. When a program assigns a value to an element of a sequence- or dictionary-like object, the object's **__setitem__** method executes.
- Python defines several special methods to provide a mapping-based interface to its clients.

TERMINOLOGY

__abs__ method (overloads built-in function **abs**)
__add__ method (overloads operator **+**)
__and__ method (overloads operator **&**)
"arity"
append
binary operator
clear
__coerce__ method (overloads coercion behavior)
__complex__ method (overloads built-in function **complex**)
__contains__ method (overloads operator **in**)
copy
count
__delattr__ method (overloads attribute deletion)
__delitem__ method (overloads sequence/mapping element deletion)
__div__ method (overloads operator **/**)
__divmod__ method (overloads built-in function **divmod**)
__float__ method (overloads built-in function **float**)
__floordiv__ method (overloads operator **//**)
get
__getattr__ method (overloads attribute retrieval)

__getitem__ method (overloads sequence/mapping element retrieval)
has_key
__hex__ method (overloads built-in function hex)
__iadd__ method (overloads symbol +=)
__iand__ method (overloads symbol &=)
__idiv__ method (overloads symbol /=)
__ifloordiv__ method (overloads symbol //=)
__ilshift__ method (overloads symbol <<=)
__imod__ method (overloads symbol %=)
__imul__ method (overloads symbol *=)
index
insert
__int__ method (overloads built-in function int)
__invert__ method (overloads operator ~)
__ior__ method (overloads symbol |=)
__ipow__ method (overloads symbol **=)
__irshift__ method (overloads symbol >>=)
__isub__ method (overloads symbol -=)
items
__ixor__ method (overloads symbol ^=)
keys
__len__ method (overloads built-in function len)
__long__ method (overloads built-in function long)
__lshift__ method (overloads operator <<)
__mod__ method (overloads operator %)
__mul__ method (overloads operator *)
__neg__ method (overloads operator -)
__oct__ method (overloads built-in function oct)
operator overloading
__or__ method (overloads operator |)
pop
__pos__ method (overloads operator +)
__pow__ method (overloads operator **)
__radd__ method (overloads right-hand addition)
__rand__ method (overloads right-hand bitwise AND)
__rdiv__ method (overloads right-hand division)
remove
__repr__ method (formal string representation)
reverse
__rfloordiv__ method (overloads right-hand floor division)
__rlshift__ method (overloads right-hand left-shift)
__rmod__ method (overloads right-hand modulus)
__rmul__ method (overloads right-hand multiplication)
__ror__ method (overloads right-hand bitwise OR)
__rpow__ method (overloads right-hand exponentiation)
__rshift__ method (overloads operator >>)
__rrshift__ method (overloads right-hand right-shift)
__rsub__ method (overloads right-hand subtraction)
__rxor__ method (overloads right-hand bitwise exclusive OR)
__setattr__ method (overloads attribute assignment)
__setitem__ method (overloads sequence/mapping element assignment)
sort
special method
__str__ method (informal string string representation)
__sub__ method (overloads operator -)
unary operator
update
values
__xor__ method (overloads operator ^)

SELF-REVIEW EXERCISES

8.1 Fill in the blanks in each of the following statements:

a) Special methods __________, __________ and __________ customize attribute access through the dot access operator.

b) Suppose a and b are integer variables and a program calculates the sum a + b. Now suppose c and d are string variables and a program performs the concatenation c + d. The

two **`+`** operators here are clearly being used for different purposes. This is an example of __________.

c) The method name __________ overloads the **`+`** operator.
d) The __________, __________ and __________ of an operator cannot be changed by overloading.
e) The **`print`** statement implicitly invokes special method __________.
f) Special method **`__coerce__`** should return __________ if no coercion can be made.
g) Special method **`__ne__`** overloads the __________.
h) Special method __________ customizes the behavior of built-in function **`abs`**.
i) Special method __________ overloads the exponentiation operator __________.
j) Special methods __________, __________ and __________ control attribute access through the **`[]`** subscript operators for list- and dictionary-like types.

8.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) Customization is accomplished by implementing special methods.
b) Python allows the programmer to create new operators to overload.
c) Overloading a mathematical operator implicitly overloads its augmented assignment counterpart.
d) User-defined objects can use Python's implicit operator overloading to get the expected results.
e) A class may overload the operation of the **`=`** assignment symbol.
f) Unary operators can be overloaded to accept two operands.
g) Operator overloading cannot change how an operator works with built-in types.
h) Comparison operators can be overloaded.
i) Subtraction can be overloaded with either special method **`__neg__`** or **`__sub__`**.
j) A class must define special methods to provide a dictionary-like interface.

ANSWERS TO SELF-REVIEW EXERCISES

8.1 a) **`__getattr__`**, **`__setattr__`**, **`__delattr__`**. b) operator overloading. c) **`__add__`**. d) precedence, associativity, "arity." e) **`__str__`**. f) **`None`**. g) **`!=`** and **`<>`** inequality operators. h) **`__abs__`**. i) **`__pow__`**, **`**`**. j) **`__getitem__`**, **`__setitem__`**, **`__delitem__`**.

8.2 a) True. b) False. Python prohibits the programmer from creating new operators. c) True. d) False. To use an operator or a statement on objects, that operator or statement must be overloaded. e) False. The assignment symbol cannot be overloaded. f) False. Unary operators can be overloaded, but the number of operands an operator takes cannot be changed. g) True. h) True. i) False. Subtraction can be overloaded only with **`__sub__`**; the unary operator **`-`** can be overloaded with method **`__neg__`**. j) False. A class may define special methods to provide a dictionary-like interface, but may also use inheritance.

EXERCISES

8.3 The definition for class **`SimpleDictionary`** in Fig. 8.15 does not include all the methods suggested for providing a dictionary interface. Review the list of mapping methods in Fig. 8.14, and modify the definition for class **`SimpleDictionary`** to include definitions for methods **`clear`**, **`copy`**, **`get`**, **`has_key`** and **`update`**. Each method of class **`SimpleDictionary`** should call attribute **`__dict__`**'s corresponding method, passing any necessary arguments. Review the description of dictionary methods in Section 5.6—the corresponding methods of class **`SimpleDictionary`** should specify the same arguments and should return the same value.

8.4 Implement methods **`append`**, **`count`**, **`index`**, **`insert`**, **`pop`**, **`remove`**, **`reverse`** and **`sort`** for class **`SingleList`**. Review the description of list methods in Section 5.6—the corre-

sponding `SingleList` methods should specify the same arguments and should return the same value. Any new method that modifies the list should ensure that only unique values are inserted. The method should raise an exception if the client attempts to insert an existing value. Also, implement methods `__delitem__` and `__contains__` to enable clients to delete list elements with keyword `del` or perform membership tests with keyword `in`.

8.5 Review the `Rational` class definition (Fig. 8.9) and driver (Fig. 8.10). What happens when Python executes the following statement?

```
x = 1 + Rational( 3, 4 )
```

Special methods `__radd__`, `__rsub__` and so on overload the mathematical operators for a user-defined class when an object of that class is used as the right-hand value of an operator. For each operator overloaded in Fig. 8.9 (i.e., operators `+`, `-`, `*`, `/` and `//`), add a corresponding method for overloading the operator when a `Rational` appears to the right of that operator.

8.6 As class `Rational` is currently implemented, the client may modify the attributes (i.e., `sign`, `numerator` and `denominator`) and place the data in an inconsistent state. Modify the definition for class `Rational` from Exercise 8.5 to include method `__setitem__`. If a client attempts to change the numerator or denominator of an object of class `Rational`, `__setitem__` determines whether the change affects the sign of the object. If so, the method changes the object's sign and sets the numerator or denominator as the absolute value of the client-specified value. The method also should call method `simplify` to reduce the object. Beware: If `__setitem__` assigns a value to an attribute through the dot access operator, Python invokes `__setitem__` again, resulting in infinite recursion. Make sure the method makes assignments through the object's `__dict__` attribute instead. [*Note:* Methods `__init__` and `simplify` also must be updated to use the object's `__dict__`, to avoid infinite recursion].

8.7 Consider a class `Complex` that simulates the built-in complex data type. The class enables operations on so-called *complex numbers*. These are numbers of the form `realPart + imaginaryPart *` *i*, where *i* has the value

$$\sqrt{-1}$$

a) Modify the class to enable output of complex numbers in the form (*realPart*, *imaginaryPart*i), through the overloaded `__str__` method.
b) Overload the multiplication operator to enable multiplication of two complex numbers as in algebra, using the equation
`(a, bi) * (c, di) = (a*c - b*d, (a*d + b*c)i)`
c) Overload the `==` operator to allow comparisons of complex numbers. [*Note:* `(a, bi)` is equal to `(c, di)` if `a` is equal to `c` and `b` is equal to `d`.]

```
# Exercise 8.7: Complex.py
# Complex number class.

class Complex:
    """Complex numbers of the form realPart + imaginaryPart * i"""

    def __init__( self, real = 0, imaginary = 0 ):
        """Assigns values to realPart and imaginaryPart"""

        self.realPart = real
        self.imaginaryPart = imaginary
```

```

   def __add__( self, other ):
      """Returns the sum of two Complex instances"""

      real = self.realPart + other.realPart
      imaginary = self.imaginaryPart + other.imaginaryPart

      # create and return new complexNumber
      return Complex( real, imaginary )

   def __sub__( self, other ):
      """Returns the difference of two Complex instance"""

      real = self.realPart - other.realPart
      imaginary = self.imaginaryPart - other.imaginaryPart

      # create and return new complexNumber
      return Complex( real, imaginary )
```

8.8 Develop class **`Polynomial`**. The internal representation of a **`Polynomial`** is a dictionary of terms. Each term is a key–value pair that contains an exponent and a coefficient. The term

$$2x^4$$

has the coefficient 2 and the exponent 4. For simplicity, assume the polynomial contains only nonnegative exponents. Develop the class with a dictionary-based interface for accessing terms that includes the following elements:

a) The class's constructor accepts a dictionary of *exponent*`:`*coefficient* pairs.
b) Coefficient values in a **`Polynomial`** are accessed by exponent keys (e.g., **`polynomial[ exponent ] = coefficient`**). If a polynomial does not have a coefficient for a specified exponent, the expression **`polynomial[ exponent ]`** evaluates to 0.
c) The length of a **`Polynomial`** is the value of its highest exponent.
d) Define method **`__str__`** for representing a Polynomial as a string with terms of the form cx^y.
e) Include an overloaded addition operator (**`+`**) to add two **`Polynomial`**s.
f) Include an overloaded subtraction operator (**`-`**) to subtract two **`Polynomial`**s.

9

Object-Oriented Programming: Inheritance

Objectives

- To create new classes by inheriting from existing classes.
- To understand how inheritance promotes software reusability.
- To understand the notions of base class and derived class.
- To understand the concept of polymorphism.
- To learn about classes that inherit from base-class **`object`**.

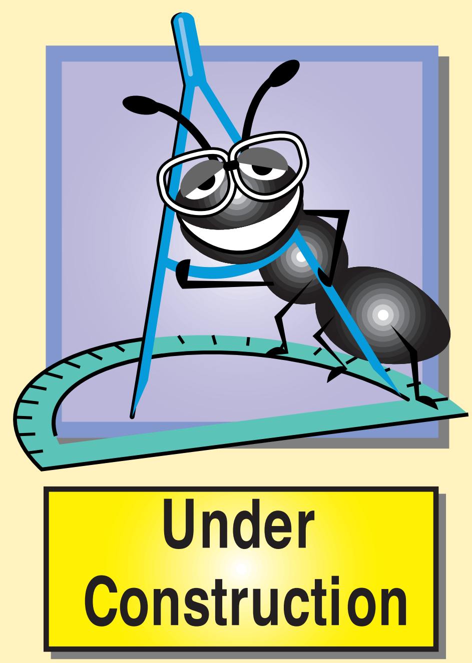
Say not you know another entirely, till you have divided an inheritance with him.
Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
Bertrand Russell

A deck of cards was built like the purest of hierarchies, with every card a master to those below it, a lackey to those above it.
Ely Culbertson

Good as it is to inherit a library, it is better to collect one.
Augustine Birrell

Save base authority from others' books.
William Shakespeare, Love's Labours Lost



Outline

9.1 Introduction

In this chapter we discuss *inheritance*—one of the most important capabilities of object-oriented programming. Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors, and overriding or embellishing these with capabilities the new classes require. Software reusability saves time in program development. It encourages programmers to reuse proven and debugged high-quality software, thus reducing problems after a system becomes functional. These are exciting possibilities.

When creating a new class, instead of writing completely new attributes and methods, the programmer can designate that the new class is to *inherit* the attributes and methods of a previously defined *base class*. The new class is referred to as a *derived class*. Each derived class itself can be a base class for some future derived class. With *single inheritance*, a class is derived from one base class. With *multiple inheritance*, a derived class inherits from several base classes. Single inheritance is straightforward—we show several examples that should enable the reader to become proficient quickly. Multiple inheritance is beyond the scope this edition—we do not show a live-code example and issue a strong caution urging the reader to pursue further study before using this powerful capability. Appendix O, Additional Python 2.2 Features, describes new Python 2.2 features that enable the programmer to exercise more control over program execution when using multiple inheritance in a more manner.

A derived class can add attributes and methods of its own, so an object of a derived class can be larger than object of that derived-class's base class. A derived class is more specific than its base class and represents a smaller set of objects. With single inheritance, the derived class starts out essentially the same as the base class. The real strength of inheritance comes from the ability to define in the derived class additions, replacements or refinements for the features inherited from the base class.

With inheritance, every object of a derived class also may be treated as an object of that derived class's base class. We can take advantage of this "derived-class-object-is-a-base-class-object" relationship to perform some interesting manipulations. For example, we can thread a wide variety of different objects related through inheritance into a list where each element of the list is treated as a base-class object. This allows a variety of objects to be processed in a general way. As we will see, this capability—called *polymorphism*—is a key thrust of object-oriented programming (OOP).

With polymorphism, it is possible to design and implement systems that are more easily *extensible.* Programs can be written to process generically—as base-class objects—objects of all existing classes in a hierarchy. Classes that do not exist during program development can be added with little or no modification to the generic part of the program—as long as those classes are part of the hierarchy that is being processed generically. The only parts of a program that need modification are those parts that require direct knowledge of the particular class that is added to the hierarchy. Polymorphism enables us to write programs in a general fashion to handle many existing and yet-to-be-specified related classes. Inheritance and polymorphism are effective techniques for managing software complexity.

Experience in building software systems indicates that significant portions of the code deal with closely related special cases. It becomes difficult in such systems to see the "big picture" because the designer and the programmer become preoccupied with the special cases. Object-oriented programming provides several ways of "seeing the forest through the trees"—a process called *abstraction*.

We distinguish between *"is-a" relationships* and *"has-a" relationships*. "Is a" is inheritance. In an "is a" relationship, an object of a derived-class type may also be treated as an object of the base-class type. "Has a" is composition (see Fig. 7.18). In a "has a" relationship, an object *has* references to one or more objects of other classes as members.

A derived class can access the attributes and methods of its base class. One problem with inheritance is that a derived class can inherit method implementations that it does not need to have or should expressly not have. When a base-class method implementation is inappropriate for a derived class, that method can be *overridden* (i.e., redefined) in the derived class with an appropriate implementation.

Perhaps most exciting is the notion that new classes can inherit from classes in existing *class libraries.* Organizations develop their own class libraries and use other libraries available worldwide. Eventually, software will be constructed predominantly from *standardized reusable components* just as hardware is often constructed today. This will help to meet the challenges of developing the ever more powerful software we will need in the future.

9.2 Inheritance: Base Classes and Derived Classes

Often an object of one class really "is an" object of another class as well. A rectangle certainly *is a* quadrilateral (as are a square, a parallelogram and a trapezoid). Thus, class **`Rectangle`** can be said to inherit from class **`Quadrilateral`**. In this context, class

`Quadrilateral` is a base class and class **`Rectangle`** is a derived class. A rectangle *is a* specific type of quadrilateral, but it is incorrect to claim that a quadrilateral *is a* rectangle (the quadrilateral could, for example, be a parallelogram). Figure 9.1 shows several simple inheritance examples.

Other object-oriented programming languages such as Smalltalk and Java use different terminology: In inheritance, the base class is called the *superclass* and the derived class is called the *subclass*. Because inheritance normally produces derived classes with *more* features than their base classes, the terms superclass and subclass can be confusing; we avoid these terms.

Inheritance forms tree-like hierarchical structures. A base class exists in a hierarchical relationship with its derived classes. A class can certainly exist by itself, but it is when a class is used with the mechanism of inheritance that the class becomes either a base class that supplies attributes and behaviors to other classes or a derived class that inherits attributes and behaviors.

Let us develop a simple inheritance hierarchy (Fig. 9.2). A typical university community has thousands of people who are community members. These people consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teaching faculty. This yields the inheritance hierarchy shown in Fig. 9.2. Note that some administrators also teach classes, so we have used multiple inheritance to form class **`AdministratorTeacher`**. Because students often work for their universities, and because employees often take courses, it would also be reasonable to use multiple inheritance to create a class called **`EmployeeStudent`**.

Another inheritance hierarchy is the **`Shape`** hierarchy of Fig. 9.3. A common observation among students learning object-oriented programming is that there are abundant examples of hierarchies in the real world. It is just that these students are not accustomed to categorizing the real world in this manner, so it takes some adjustment in their thinking.

Base class	Derived classes
`Student`	**`GraduateStudent`** **`UndergraduateStudent`**
`Shape`	**`Circle`** **`Triangle`** **`Rectangle`**
`Loan`	**`CarLoan`** **`HomeImprovementLoan`** **`MortgageLoan`**
`Employee`	**`FacultyMember`** **`StaffMember`**
`Account`	**`CheckingAccount`** **`SavingsAccount`**

Fig. 9.1 Inheritance examples.

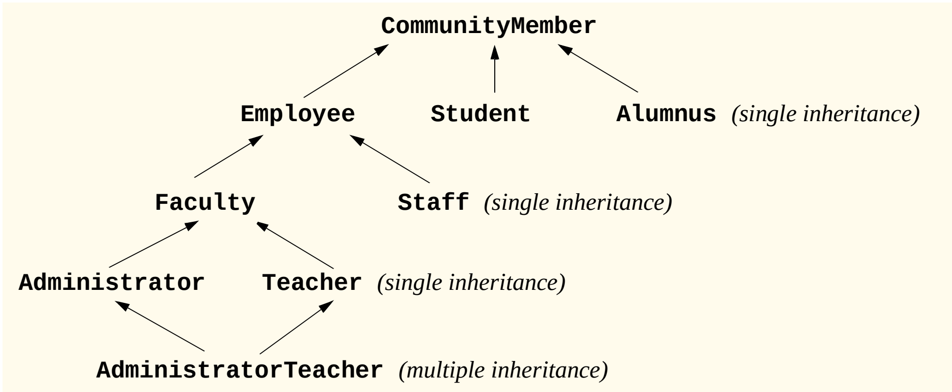


Fig. 9.2 Inheritance hierarchy for university community members.

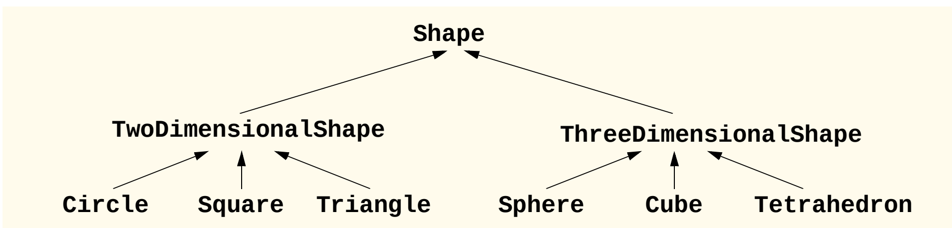


Fig. 9.3 **`Shape`** class hierarchy.

Let us consider the syntax for indicating inheritance. To specify that class **`TwoDimensionalShape`** is derived from class **`Shape`**, class **`TwoDimensionalShape`** typically would be defined as follows:

```
class TwoDimensionalShape( Shape ):
      ...
```

With inheritance, all attributes and methods of the base class are inherited as attributes and methods of the derived class.

A base class may be either a *direct base class* of a derived class or an *indirect base class* of a derived class. A direct base class of a derived class is explicitly listed inside parentheses (**`()`**) when the derived class is defined. An indirect base class is not explicitly listed when the derived class is defined; rather, the indirect base class is inherited from two or more levels up the class hierarchy. In Fig. 9.3, class **`Circle`** has a direct base class **`TwoDimensionalShape`** and an indirect base class **`Shape`**. Although the class definition for class **`Circle`** would list only class **`TwoDimensionalShape`** as a base class, class **`Circle`** would inherit all the attributes and methods of class **`TwoDimensionalShape`** and of class **`Shape`**.

It is possible to treat base-class objects and derived-class objects similarly; that commonality is expressed in the attributes and behaviors of the base class. Objects of any class derived with inheritance from a common base class can all be treated as objects of that base class. In Section 9.10, we consider an example in which we can take advantage of this relationship.

9.3 Creating Base Classes and Derived Classes

This section creates an inheritance hierarchy and instantiates objects from the classes in that hierarchy. Python provides two built-in functions—***issubclass*** and ***isinstance***—that enable us to determine whether one class is derived from another class and whether a value is an object of a particular class or of a subclass of that class. We discuss these functions in Fig. 9.4 that demonstrates how to derive one class from another class and that underscores the fact that a derived-class object "is a" base-class object. In Fig. 9.4, lines 6–13 show a **Point** class and constructor definition. Lines 15–28 show a **Circle** class and method definitions. Lines 30–52 contain a driver program.We offer this hierarchy as an example of so-called *structural inheritance*. Although it may not appear to be a natural series of "is-a" relationships (i.e., many readers will be uncomfortable with any claim that a circle is a point), the fact that we derive **Circle** from **Point** makes a **Circle** a **Point** in a mechanical sense. We find that this example helps the student understand the mechanics of inheritance. Later in the chapter, we present a natural example of inheritance.

```
1  # Fig 9.4: fig09_04.py
2  # Derived class inheriting from a base class.
3
4  import math
5
6  class Point:
7     """Class that represents geometric point"""
8
9     def __init__( self, xValue = 0, yValue = 0 ):
10       """Point constructor takes x and y coordinates"""
11
12       self.x = xValue
13       self.y = yValue
14
15 class Circle( Point ):
16    """Class that represents a circle"""
17
18    def __init__( self, x = 0, y = 0, radiusValue = 0.0 ):
19       """Circle constructor takes x and y coordinates of center
20       point and radius"""
21
22       Point.__init__( self, x, y )  # call base-class constructor
23       self.radius = float( radiusValue )
24
25    def area( self ):
26       """Computes area of a Circle"""
27
28       return math.pi * self.radius ** 2
29
30 # main program
31
32 # examine classes Point and Circle
33 print "Point bases:", Point.__bases__
34 print "Circle bases:", Circle.__bases__
```

Fig. 9.4 Derived class inheriting from a base class. (Part 1 of 2.)

```
35
36   # demonstrate class relationships with built-in function issubclass
37   print "\nCircle is a subclass of Point:", \
38      issubclass( Circle, Point )
39   print "Point is a subclass of Circle:", issubclass( Point, Circle )
40
41   point = Point( 30, 50 )  # create Point object
42   circle = Circle( 120, 89, 2.7 )   # create Circle object
43
44   # demonstrate object relationship with built-in function isinstance
45   print "\ncircle is a Point object:", isinstance( circle, Point )
46   print "point is a Circle object:", isinstance( point, Circle )
47
48   # print Point and Circle objects
49   print "\npoint members:\n\t", point.__dict__
50   print "circle members:\n\t", circle.__dict__
51
52   print "\nArea of circle:", circle.area()
```

```
Point bases: ()
Circle bases: (<class __main__.Point at 0x00767250>,)

Circle is a subclass of Point: 1
Point is a subclass of Circle: 0

circle is a Point object: 1
point is a Circle object: 0

point members:
        {'y': 50, 'x': 30}
circle members:
        {'y': 89, 'x': 120, 'radius': 2.7000000000000002}

Area of circle: 22.9022104447
```

Fig. 9.4 Derived class inheriting from a base class. (Part 2 of 2.)

The constructor for class **Point** (lines 9–13) takes two arguments that correspond to the point's *x*- and *y*-coordinates. Class **Circle** (lines 15–28) inherits from class **Point**. The parentheses (**()**) in the first line of the class definition indicate inheritance. The name of the base class (**Point**) is placed inside the parentheses. Class **Circle** inherits all attributes of class **Point**. This means that class **Circle** contains the **Point** members (i.e., **x** and **y**) as well as the **Circle** members.

A derived class inherits the methods defined in its base class, including the base-class constructor. Often, the derived class *overrides* the base-class constructor by defining a derived-class **__init__** method. A derived class overrides a base-class method when the derived class defines a method with the same name as a base-class method. The overridden derived-class constructor usually calls the base-class constructor, to initialize base-class attributes before initializing derived-class attributes. Line 22 in the **Circle** constructor calls the base-class constructor through an *unbound method call*. Until now, we have invoked only *bound method calls*. A bound method call is invoked by accessing the method

name through an object (e.g., **anObject.method()**). We have seen that Python inserts the object-reference argument for bound method calls. An unbound method call is invoked by accessing the method through its class name and specifically passing an object reference. For example, line 22 calls method **Point.__init__** and passes **self** (an object of class **Circle**) as the object reference. The unbound method call also passes the values for **x** and **y** so the **Point** constructor can initialize the **Point** attributes for the object of class **Circle**. We explore method overriding and bound and unbound method calls further in the next section. After the base-class constructor terminates, control returns to the **Circle** constructor so it can perform any **Circle**-specific initialization. Line 23 adds a new attribute—**radius**—to **Circle**'s namespace.

Software Engineering Observation 9.1

A derived class (like any class) is not required to define a constructor. If a derived class does not define a constructor, the class's base-class constructor executes when the client creates a new object of the class.

Common Programming Error 9.1

If a derived class's overridden constructor needs to invoke the base-class constructor to initialize base-class members, the derived-class constructor must invoke the base-class constructor explicitly. Failure to call the base-class constructor from a derived class often is a logic error.

Common Programming Error 9.2

Failure to specify an object reference as the first argument to an unbound method call is a logic error.

Lines 25–28 define method **area** for class **Circle**. This method demonstrates how the derived class can define new methods to extend the functionality of the base class. In this example, derived class **Circle** provides extra functionality that computes the area of an object of class **Circle**.

The driver program in Fig. 9.4 first prints the value of each class's **__bases__** attribute (lines 33–34). Recall from Chapter 7 that each class contains special attributes, including **__bases__**, which is a tuple that contains references to each of the class's base classes. Notice from the output that **Point.__bases__** is an empty tuple, because **Point** does not inherit from any other class. However, **Circle.__bases__** is a tuple that contains one value—a reference to base-class **Point**. Lines 37–39 call built-in function **issubclass** to demonstrate that **Circle** is a subclass of **Point**, but that **Point** is not a subclass of **Circle**. Function **issubclass** takes two arguments that are classes and returns true if the first argument is a class that inherits from the second argument (or if the first argument is the same class as the second argument).

Lines 41–42 create **point** as a reference to an object of class **Point** and **circle** as a reference to an object of class **Circle**. Lines 45–46 demonstrate built-in function **isinstance**. This function takes two arguments—an object and a class. If the object argument is an object of the type specified by the class argument, or if the object argument is an object of a derived class of the type specified by the class argument, function **isinstance** returns 1. Otherwise, the function returns 0. The two calls to function **isinstance** demonstrate that a derived class is an object of its base class (e.g., **circle** is a **Point**), but the reverse is not true (e.g., **point** is not a **Circle**).

Common Programming Error 9.3

Treating a base-class object as a derived-class object can cause runtime errors. A program terminates if the program attempts to call a derived-class method from a base-class object and the base class does not define that method.

Lines 49–50 print the `__dict__` attribute `point` and `circle`, respectively. Notice from the output that `circle`'s `__dict__` contains attributes `x` and `y`, initialized in the base-class constructor. Line 52 calls `circle` method `area`, to demonstrate class `Circle`'s extended functionality.

In this section, we demonstrated the mechanics of defining base and derived classes and discussed bound and unbound methods. This material establishes the foundation we need for our deeper treatment of object-oriented programming with inheritance in the remainder of this chapter.

9.4 Overriding Base-Class Methods in a Derived Class

A derived class can override a base-class method by supplying a new version of that method with the same name. When that method is mentioned by name in the derived class, the derived-class version is selected. The name of the base class may be used to access the base-class version from the derived class by passing the derived-class object in an unbound call to the base-class's method.

Common Programming Error 9.4

When a base-class method is overridden in a derived class, it is common to have the derived-class version call the base-class version and perform some additional work. Not using the base-class name to reference (i.e., prepending the base-class name and a dot to) the base-class method causes infinite recursion, because the derived-class method actually calls itself. This eventually will cause the system to exhaust memory—a fatal error.

Consider a simplified class `Employee`. It stores the employee's `firstName` and `lastName`. This information is common to all employees, including classes derived from class `Employee`. From class `Employee`, now derive classes `HourlyWorker`, `PieceWorker`, `Boss` and `CommissionWorker`. The `HourlyWorker` gets paid by the hour, with "time-and-a-half" for overtime hours in excess of 40 hours per week. The `PieceWorker` gets paid a fixed rate per item produced—for simplicity, assume this person makes only one type of item, so the data members are number of items produced and rate per item. The `Boss` gets a fixed wage per week. The `CommissionWorker` gets a small fixed weekly base salary plus a fixed percentage of that person's gross sales for the week. For simplicity, this and the next section present only class `Employee` and derived class `HourlyWorker`. In Section 9.10, we present a case study that addresses the entire hierarchy.

Our next example appears in Fig. 9.5. Lines 4–16 show the `Employee` class definition and `Employee` methods. Lines 18–40 show the `HourlyWorker` class definition and `HourlyWorker` method definitions. Lines 42–49 show a driver program for the `Employee`/`HourlyWorker` inheritance hierarchy that creates an object of class `HourlyWorker` and invokes its `__str__` method implicitly, then explicitly with a bound method call, then explicitly with an unbound method call.

The `Employee` class definition consists of two attributes (`firstName` and `lastName`) and two methods (`__init__` and `__str__`). The constructor receives two arguments and assigns their values to `firstName` and `lastName`. Class `HourlyWorker`

inherits from class **Employee**. The members of **HourlyWorker** include attributes **hours** and **wage** and methods **__init__**, **getPay** and **__str__**.

```
# Fig. 9.5: fig09_05.py
# Overriding base-class methods.

class Employee:
   """Class to represent an employee"""

   def __init__( self, first, last ):
      """Employee constructor takes first and last name"""

      self.firstName = first
      self.lastName = last

   def __str__( self ):
      """String representation of an Employee"""

      return "%s %s" % ( self.firstName, self.lastName )

class HourlyWorker( Employee ):
   """Class to represent an employee paid by hour"""

   def __init__( self, first, last, initHours, initWage ):
      """Constructor for HourlyWorker, takes first and last name,
      initial number of hours and initial wage"""

      Employee.__init__( self, first, last )
      self.hours = float( initHours )
      self.wage = float( initWage )

   def getPay( self ):
      """Calculates HourlyWorker's weekly pay"""

      return self.hours * self.wage

   def __str__( self ):
      """String representation of HourlyWorker"""

      print "HourlyWorker.__str__ is executing"""

      return "%s is an hourly worker with pay of $%.2f" % \
         ( Employee.__str__( self ), self.getPay() )

# main program
hourly = HourlyWorker( "Bob", "Smith", 40.0, 10.00 )

# invoke __str__ method several ways
print "Calling __str__ several ways..."
print hourly  # invoke __str__ implicitly
print hourly.__str__()  # invoke __str__ explicitly
print HourlyWorker.__str__( hourly )  # explicit, unbound call
```

Fig. 9.5 Overriding base-class methods in a derived class. (Part 1 of 2.)

```
Calling __str__ several ways...
 HourlyWorker.__str__ is executing
Bob Smith is an hourly worker with pay of $400.00
HourlyWorker.__str__ is executing
Bob Smith is an hourly worker with pay of $400.00
HourlyWorker.__str__ is executing
Bob Smith is an hourly worker with pay of $400.00
```

Fig. 9.5 Overriding base-class methods in a derived class. (Part 2 of 2.)

The **HourlyWorker** constructor uses an unbound method call to pass the strings **first** and **last** to the **Employee** constructor so the base-class attributes can be initialized, then initializes attributes **hours** and **wage**. Method **getPay** uses attributes **hours** and **wage** to calculate the salary of the **HourlyWorker**.

HourlyWorker method **__str__** overrides the **Employee** **__str__** method. Often, base-class methods are overridden in a derived class to provide more functionality. The overridden method sometimes calls the base-class version of the method to perform part of the new task. In this example, the derived-class **__str__** method calls the base-class **__str__** method (with an unbound method call on line 40) to output the employee's name. The derived-class **__str__** method also outputs the employee's pay.

The driver program invokes an hourly object's **__str__** method in three different ways. Line 47 simply uses the object in a **print** statement, which implicitly invokes the object's **__str__** method. Line 48 makes an explicit, bound call to the object's **__str__** method. Line 49 makes an unbound call to class **HourlyWorker**'s **__str__** method and passes **hourly** as the object reference argument.

9.5 Software Engineering with Inheritance

We can use inheritance to customize existing software. We inherit the attributes and behaviors of an existing class, then add attributes and behaviors (or override base-class behaviors) to customize the class to meet our needs. It can be difficult for students to appreciate the problems faced by designers and implementors on large-scale software projects. People experienced on such projects will invariably state that a key to improving the software development process is software reuse. Object-oriented programming in general, and Python in particular, certainly do this.

The availability of substantial and useful modules delivers the maximum benefits of software reuse through inheritance. As interest in Python grows, interest in creating useful modules also grows. Just as shrink-wrapped software produced by independent software vendors became an explosive growth industry with the arrival of the personal computer, so, too, is the creation and distribution of class libraries. Application designers build their applications with these libraries, and library designers are being rewarded by having their libraries wrapped with the applications.

Software Engineering Observation 9.2

Creating a derived class does not affect its base class's source code; the integrity of a base class is preserved by inheritance.

A base class specifies commonality—all classes derived from a base class inherit the capabilities of that base class. In the object-oriented design process, the designer looks for commonality and "factors it out" to form desirable base classes. Derived classes are then customized beyond the capabilities inherited from the base class.

Software Engineering Observation 9.3

In an object-oriented system, classes often are closely related. "Factor out" common attributes and behaviors and place these in a base class. Then use inheritance to form derived classes.

Just as the designer of non-object-oriented systems seeks to avoid unnecessary proliferation of functions, the designer of object-oriented systems should avoid unnecessary proliferation of classes. Such a proliferation of classes creates management problems and can hinder software reusability, simply because it is more difficult for a potential reuser of a class to locate that class in a large collection. The trade-off is to create fewer classes, each providing substantial additional functionality, but such classes might be too rich for certain users.

Performance Tip 9.1

If classes produced through inheritance are larger than they need to be, memory and processing resources may be wasted. Inherit from the class "closest" to what you need.

Note that reading a set of derived-class definitions can be confusing because inherited members are not shown, but they are nevertheless present in the derived classes. A similar problem can exist in the documentation of derived classes.

Software Engineering Observation 9.4

A derived class contains the attributes and behaviors of its base class. A derived class can also contain additional attributes and behaviors.

Software Engineering Observation 9.5

Modifications to a base class do not require derived classes to change as long as the interfaces to the base class remain unchanged.

9.6 Composition vs. Inheritance

We have discussed *is-a* relationships, which are supported by inheritance. We have also discussed *has-a* relationships (and seen an example in Chapter 7, Object-Based Programming) in which a class may have references to other classes as members. "hassuch relationships create new classes by *composition* of existing classes. For example, given the classes **`Employee`**, **`BirthDate`** and **`TelephoneNumber`**, it is improper to say that an **`Employee`** *is a* **`BirthDate`** or that an **`Employee`** *is a* **`TelephoneNumber`**. But it is certainly appropriate to say that an **`Employee`** *has a* **`BirthDate`** and that an **`Employee`** *has a* **`TelephoneNumber`**.

Software Engineering Observation 9.6

Program modifications to a class that is a member of another class do not require the enclosing class to change as long as the interface to the member class remains unchanged.

9.7 "Uses A" and "Knows A" Relationships

Inheritance and composition each encourage software reuse by creating new classes that have much in common with existing classes. There are other ways to use the services of

classes. Although a person object is not a car and a person object does not contain a car, a person object certainly *uses a* car. A program uses an object simply by calling a method of that object through a reference.

An object can be *aware of* another object. Knowledge networks frequently have such relationships. One object can contain a reference to another object to be aware of that object. In this case, one object is said to have a *knows a* relationship with the other object; this is sometimes called an *association*.

9.8 Case Study: Point, Circle, Cylinder

Consider a more substantial example using a point, circle, cylinder structural-inheritance hierarchy. First we develop and use class **`Point`** (Fig. 9.6). Then we present an example in which we derive class **`Circle`** from class **`Point`** (Fig. 9.7). Finally, we present an example in which we derive class **`Cylinder`** from class **`Circle`** (Fig. 9.8).

Figure 9.6 shows class **`Point`**. The constructor (lines 7–11) takes two arguments that correspond to the *x*- and *y*-coordinates of the point. Method **`__str__`** (lines 13–16) creates a string representation of an object of class **`Point`**. The driver program in function **`main`** (lines 19–30) creates an object of class **`point`**, prints its **`x`** and **`y`** attributes, changes the value of its attributes and prints the changed **`point`** object.

```
1  # Fig 9.6: PointModule.py
2  # Definition and test function for class Point.
3
4  class Point:
5     """Class that represents a geometric point"""
6
7     def __init__( self, xValue = 0, yValue = 0 ):
8        """Point constructor takes x and y coordinates"""
9
10       self.x = xValue
11       self.y = yValue
12
13    def __str__( self ):
14       """String representation of a Point"""
15
16       return "( %d, %d )" % ( self.x, self.y )
17
18 # main program
19 def main():
20    point = Point( 72, 115 )  # create object of class Point
21
22    # print point attributes
23    print "X coordinate is:", point.x
24    print "Y coordinate is:", point.y
25
26    # change point attributes and output new location
27    point.x = 10
28    point.y = 10
29
30    print "The new location of point is:", point
```

Fig. 9.6 Class **`Point`**—**`PointModule.py`**. (Part 1 of 2.)

```

if __name__ == "__main__":
   main()
```

```
X coordinate is: 72
Y coordinate is: 115
The new location of point is: ( 10, 10 )
```

Fig. 9.6 Class **Point**—**PointModule.py**. (Part 2 of 2.)

Figure 9.7 demonstrates class **Circle**, which inherits from class **Point** (Fig. 9.6). Lines 7–26 show the **Circle** class definition, and lines 29–45 contain the driver program for class **Circle**. Note that, because class **Circle** inherits from class **Point**, the interface to **Circle** includes the **Point** methods as well as the **Circle** method **area**.

```
1  # Fig. 9.7: CircleModule.py
2  # Definition and test function for class Circle.
3
4  import math
5  from PointModule import Point
6
7  class Circle( Point ):
8     """Class that represents a circle"""
9
10    def __init__( self, x = 0, y = 0, radiusValue = 0.0 ):
11       """Circle constructor takes center point and radius"""
12
13       Point.__init__( self, x, y )  # call base-class constructor
14       self.radius = float( radiusValue )
15
16    def area( self ):
17       """Computes area of a Circle"""
18
19       return math.pi * self.radius ** 2
20
21    def __str__( self ):
22       """String representation of a Circle"""
23
24       # call base-class __str__ method
25       return "Center = %s Radius = %f" % \
26          ( Point.__str__( self ), self.radius )
27
28 # main program
29 def main():
30    circle = Circle( 37, 43, 2.5 )  # create Circle object
31
32    # print circle attributes
33    print "X coordinate is:", circle.x
34    print "Y coordinate is:", circle.y
35    print "Radius is:", circle.radius
```

Fig. 9.7 Class **Circle**—**CircleModule.py**. (Part 1 of 2.)

```
36
37     # change circle attributes and print new values
38     circle.radius = 4.25
39     circle.x = 2
40     circle.y = 2
41
42     print "\nThe new location and radius of circle are:", circle
43     print "The area of circle is: %.2f" % circle.area()
44
45     print "\ncircle printed as a Point is:", Point.__str__( circle )
46
47  if __name__ == "__main__":
48     main()
```

```
X coordinate is: 37
Y coordinate is: 43
Radius is: 2.5

The new location and radius of circle are: Center = ( 2, 2 ) Radius =
4.250000
The area of circle is: 56.75

circle printed as a Point is: ( 2, 2 )
```

Fig. 9.7 Class **`Circle`**—**`CircleModule.py`**. (Part 2 of 2.)

The driver program creates an object of class **`Circle`**, then prints the attributes of the object. The driver program then changes the values of the object's attributes and prints the changed object. Line 43 calls **`circle`** method **`area`** to display the object's area. Finally, line 45 calls **`Point`** method **`__str__`** as an unbound method and passes **`circle`** as the object reference. This call prints the object of class **`Circle`** as an object of class **`Point`**, demonstrating how a derived-class object can be used as a base-class object.

Our last example reuses the **`Point`** and **`Circle`** class definitions from Fig. 9.6 and Fig. 9.7. Lines 8–32 show the **`Cylinder`** class definition, and lines 35–61 are the driver program for class **`Cylinder`**. Note that class **`Cylinder`** inherits from class **`Circle`**, so the interface to **`Cylinder`** includes the **`Circle`** methods and **`Point`** methods as well as the **`Cylinder`** methods **`area`** (overridden from **`Circle`**) and **`volume`**. Note that the **`Cylinder`** constructor invokes the constructor for its direct base class **`Circle`**, but not its indirect base class **`Point`**. Each derived-class constructor is responsible only for calling the constructors of that class's immediate base class.

The driver program creates an object of class **`Cylinder`** (line 38), then prints the values of the object's attributes (lines 41–44). The driver program then changes the values of the height, radius and coordinates of the cylinder (lines 47–49) and outputs the results of the changes (lines 50–51). Finally, the program makes unbound method calls to the **`Point`** and **`Circle`** **`__str__`** methods (lines 57 and 61) to print the object of class **`Cylinder`** as an object of classes **`Point`** and **`Circle`**, respectively.

This example nicely demonstrates inheritance. The reader should now be confident with the basics of inheritance. In the remainder of the chapter, we show how to program with inheritance hierarchies in a general manner.

```python
# Fig. 9.8: CylinderModule.py
# Definition and test function for class Cylinder.

import math
from PointModule import Point
from CircleModule import Circle

class Cylinder( Circle ):
   """Class that represents a cylinder"""

   def __init__( self, x, y, radius, height ):
      """Constructor for Cylinder takes x, y, height and radius"""

      Circle.__init__( self, x, y, radius )
      self.height = float( height )

   def area( self ):
      """Calculates (surface) area of a Cylinder"""

      return 2 * Circle.area( self ) + \
         2 * math.pi * self.radius * self.height

   def volume( self ):
      """Calculates volume of a Cylinder"""

      return Circle.area( self ) * height

   def __str__( self ):
      """String representation of a Cylinder"""

      return "%s; Height = %f" % \
         ( Circle.__str__( self ), self.height )

# main program
def main():

   # create object of class Cylinder
   cylinder = Cylinder( 12, 23, 2.5, 5.7 )

   # print Cylinder attributes
   print "X coordinate is:", cylinder.x
   print "Y coordinate is:", cylinder.y
   print "Radius is:", cylinder.radius
   print "Height is:", cylinder.height

   # change Cylinder attributes
   cylinder.height = 10
   cylinder.radius = 4.25
   cylinder.x, cylinder.y = 2, 2
   print "\nThe new points, radius and height of cylinder are:", \
      cylinder

   print "\nThe area of cylinder is: %.2f" % cylinder.area()
```

Fig. 9.8 Class `Cylinder`—`CylinderModule.py`. (Part 1 of 2.)

```

   # display the Cylinder as a Point
   print "\ncylinder printed as a Point is:", \
      Point.__str__( cylinder )

   # display the Cylinder as a Circle
   print "\ncylinder printed as a Circle is:", \
      Circle.__str__( cylinder )

if __name__ == "__main__":
   main()
```

```
X coordinate is: 12
Y coordinate is: 23
Radius is: 2.5
Height is: 5.7

The new points, radius and height of cylinder are: Center = ( 2, 2 )
Radius = 4.250000; Height = 10.000000

The area of cylinder is: 380.53

cylinder printed as a Point is: ( 2, 2 )

cylinder printed as a Circle is: Center = ( 2, 2 ) Radius = 4.250000
```

Fig. 9.8 Class **Cylinder**—**CylinderModule.py**. (Part 2 of 2.)

9.9 Abstract Base Classes and Concrete Classes

When we think of a class as a type, we assume that objects of that type will be created. However, there are cases in which it is useful to define classes for which the programmer never intends to create any objects. Such classes are called *abstract classes*. Because these are used as base classes in inheritance situations, we normally refer to them as *abstract base classes*.

We do not create objects of abstract classes. The sole purpose of an abstract class is to provide an appropriate base class from which classes may inherit interface and possibly implementation. Classes from which objects can be created are called *concrete classes*.

We could have an abstract base class **TwoDimensionalShape** and derive concrete classes, such as **Square**, **Circle** and **Triangle**. We could also have an abstract base class **ThreeDimensionalShape** and derive concrete classes such as **Cube**, **Sphere** and **Cylinder**. Abstract base classes are too generic to define real objects; we need to be more specific before we can think of creating objects. That is what concrete classes do; they provide the specifics that make it reasonable to create objects.

A hierarchy need not contain any abstract classes; but, as we will see, many good object-oriented systems have class hierarchies headed by an abstract base class. In some cases, abstract classes constitute the top few levels of the hierarchy. A good example of this is a shape hierarchy. The hierarchy could be headed by abstract base class **Shape**. On the next level down, we can have two more abstract base classes, namely **TwoDimensionalShape** and **ThreeDimensionalShape**. The next level down would start defining concrete

classes for two-dimensional shapes such as circles and squares and concrete classes for three-dimensional shapes such as spheres and cubes.

9.10 Case Study: Inheriting Interface and Implementation

Our next example reexamines the **Employee** hierarchy introduced in Section 9.4. This time, we implement the entire class hierarchy, heading it with abstract base class **Employee**. The derived classes of **Employee** are **Boss**, who gets paid a fixed weekly salary regardless of the number of hours worked; **CommissionWorker**, who gets a flat base salary plus a percentage of sales; **PieceWorker**, who gets paid by the number of items produced; and **HourlyWorker**, who gets paid by the hour and receives "time-and-a-half" overtime pay for hours worked in excess of 40 hours.

Each concrete **Employee** class defines method **earnings**. An **earnings** method call certainly applies generically to all employees. However, the earnings calculation for each employee differs based on the class of the employee. These classes are all derived from the base class **Employee**, so each derived class provides appropriate implementations of **earnings**. To calculate any employee's earnings, the program simply invokes the **earnings** method on that employee's object.

Let us now consider the example (Fig. 9.9). We begin with class **Employee** (lines 4–35). The methods include a constructor that takes the first name and last name as arguments; an **__str__** method; a utility method **_checkPositive** that ensures an attribute is initialized with a positive value and an abstract method **earnings**. Method **earnings** simply raises a **NotImplementedError** exception when called. [*Note*: We discuss exceptions in Chapter 12, Exception Handling.] Why does **earnings** raise an exception? The answer is that it does not make sense to provide an implementation of this method in the **Employee** class. We cannot calculate the earnings for a generic employee—we first must know the type of employee to perform a proper earnings calculation. By raising an exception in the body of the method, we ensure that each class that inherits from **Employee** must override method **earnings** with a more specific definition. The programmer never intends to call this method on an object of abstract base class **Employee**. If a derived class neglects to override **earnings** with an appropriate definition, the abstract method in the base class raises an exception when the program attempts to call **earnings** from the derived class. Similar to **earnings**, the **Employee** constructor raises an exception if a program attempts to create an object of the abstract base class. Lines 11–13 determine whether **self** is an object of class **Employee** and, if so, raise an appropriate exception.

Class **Boss** (lines 37–54) derives from class **Employee**. The **Boss**'s methods include a constructor (lines 40–44), the overridden **earnings** method (lines 46–49) and an **__str__** method (lines 51–54). The constructor (method **__init__**) takes a first name, a last name and a weekly salary as arguments and passes the first and last names to the **Employee** constructor to initialize the **firstName** and **lastName** members of the base-class part of the derived-class object. Method **earnings** performs the **Boss**-specific earnings calculations. Method **__str__** creates a string with the type and name of the employee.

Class **CommissionWorker** (lines 56–77) derives from class **Employee**. The methods include a constructor (lines 59–66), the overridden **earnings** method (lines 68–71) and an **__str__** method (lines 73–77). The constructor takes a first name, a last name, a salary, a commission and a quantity of items sold as arguments and passes the first and last names to the **Employee** constructor. Method **earnings** performs the

CommissionWorker-specific earnings calculations. Method **__str__** creates a string with the type and name of the employee.

```
# Fig 9.9: fig09_09.py
# Creating a class hierarchy with an abstract base class.

class Employee:
   """Abstract base class Employee"""

   def __init__( self, first, last ):
      """Employee constructor, takes first name and last name.
      NOTE: Cannot create object of class Employee."""

      if self.__class__ == Employee:
         raise NotImplementedError, \
            "Cannot create object of class Employee"

      self.firstName = first
      self.lastName = last

   def __str__( self ):
      """String representation of Employee"""

      return "%s %s" % ( self.firstName, self.lastName )

   def _checkPositive( self, value ):
      """Utility method to ensure a value is positive"""

      if value < 0:
         raise ValueError, \
            "Attribute value (%s) must be positive" % value
      else:
         return value

   def earnings( self ):
      """Abstract method; derived classes must override"""

      raise NotImplementedError, "Cannot call abstract method"

class Boss( Employee ):
   """Boss class, inherits from Employee"""

   def __init__( self, first, last, salary ):
      """Boss constructor, takes first and last names and salary"""

      Employee.__init__( self, first, last )
      self.weeklySalary = self._checkPositive( float( salary ) )

   def earnings( self ):
      """Compute the Boss's pay"""

      return self.weeklySalary
```

Fig. 9.9 Abstract class-based hierarchy. (Part 1 of 3.)

```

   def __str__( self ):
      """String representation of Boss"""

      return "%17s: %s" % ( "Boss", Employee.__str__( self ) )

class CommissionWorker( Employee ):
   """CommissionWorker class, inherits from Employee"""

   def __init__( self, first, last, salary, commission, quantity ):
      """CommissionWorker constructor, takes first and last names,
      salary, commission and quantity"""

      Employee.__init__( self, first, last )
      self.salary = self._checkPositive( float( salary ) )
      self.commission = self._checkPositive( float( commission ) )
      self.quantity = self._checkPositive( quantity )

   def earnings( self ):
      """Compute the CommissionWorker's pay"""

      return self.salary + self.commission * self.quantity

   def __str__( self ):
      """String representation of CommissionWorker"""

      return "%17s: %s" % ( "Commission Worker",
         Employee.__str__( self ) )

class PieceWorker( Employee ):
   """PieceWorker class, inherits from Employee"""

   def __init__( self, first, last, wage, quantity ):
      """PieceWorker constructor, takes first and last names, wage
      per piece and quantity"""

      Employee.__init__( self, first, last )
      self.wagePerPiece = self._checkPositive( float( wage ) )
      self.quantity = self._checkPositive( quantity )

   def earnings( self ):
      """Compute PieceWorker's pay"""

      return self.quantity * self.wagePerPiece

   def __str__( self ):
      """String representation of PieceWorker"""

      return "%17s: %s" % ( "Piece Worker",
         Employee.__str__( self) )

class HourlyWorker( Employee ):
   """HourlyWorker class, inherits from Employee"""
```

Fig. 9.9 Abstract class-based hierarchy. (Part 2 of 3.)

```
103
104    def __init__( self, first, last, wage, hours ):
105       """HourlyWorker constructor, takes first and last names,
106       wage per hour and hours worked"""
107
108       Employee.__init__( self, first, last )
109       self.wage = self._checkPositive( float( wage ) )
110       self.hours = self._checkPositive( float( hours ) )
111
112    def earnings( self ):
113       """Compute HourlyWorker's pay"""
114
115       if self.hours <= 40:
116          return self.wage * self.hours
117       else:
118          return 40 * self.wage + ( self.hours - 40 ) *\
119            self.wage * 1.5
120
121    def __str__( self ):
122       """String representation of HourlyWorker"""
123
124       return "%17s: %s" % ( "Hourly Worker",
125          Employee.__str__( self ) )
126
127 # main program
128
129 # create list of Employees
130 employees = [ Boss( "John", "Smith", 800.00 ),
131               CommissionWorker( "Sue", "Jones", 200.0, 3.0, 150 ),
132               PieceWorker( "Bob", "Lewis", 2.5, 200 ),
133               HourlyWorker( "Karen", "Price", 13.75, 40 ) ]
134
135 # print Employee and compute earnings
136 for employee in employees:
137    print "%s earned $%.2f" % ( employee, employee.earnings() )
```

```
             Boss: John Smith earned $800.00
Commission Worker: Sue Jones earned $650.00
     Piece Worker: Bob Lewis earned $500.00
    Hourly Worker: Karen Price earned $550.00
```

Fig. 9.9 Abstract class-based hierarchy. (Part 3 of 3.)

Class **PieceWorker** (lines 79–99) derives from class **Employee**. The methods include a constructor (lines 82–88), the overridden **earnings** method (lines 90–93), and an **__str__** method (lines 95–99). The constructor takes a first name, a last name, a wage per piece and a quantity of items produced as arguments and passes the first and last names to the **Employee** constructor. Method **earnings** performs the **PieceWorker**-specific earnings calculations. Method **__str__** method creates a string with the type and name of the employee.

Class **HourlyWorker** (lines 101–125) derives from class **Employee**. The methods include a constructor (lines 104–110), the overridden **earnings** method (lines 112–119), and an **__str__** method (lines 121–125). The constructor takes a first name, a last name, a wage and the number of hours worked as arguments and passes the first and last names to the **Employee** constructor. Method **earnings** performs the **HourlyWorker**-specific earnings calculations.

The driver program is shown in lines 127–137. We create a list of four concrete objects of class **Employee**—an object of class **Boss**, an object of class **CommissionWorker**, an object of class **PieceWorker** and an object of class **HourlyWorker**. Lines 136–137 iterate over the list of objects of class **Employee** and call method **earnings** for each object in the list. This technique—generically processing a list of objects of various classes—is possible because of Python's inherent *polymorphic* behavior, a topic we discuss in the next section.

9.11 Polymorphism

Python enables *polymorphism*—the ability for objects of different classes related by inheritance to respond differently to the same message (i.e., method call). The same message sent to many different types of objects takes on "many forms"—hence the term polymorphism. If, for example, class **Rectangle** is derived from class **Quadrilateral**, then a **Rectangle** *is a* more specific version of a **Quadrilateral**. An operation (such as calculating the perimeter or the area) that can be performed on an object of class **Quadrilateral** also can be performed on an object of class **Rectangle**. Python is inherently polymorphic because the language is "dynamically typed." This means that Python determines at runtime whether an object defines a method or contains an attribute. If so, Python calls the appropriate method or accesses the appropriate attribute. Also, Python's dynamic typing enables programs to perform generic processing on objects of classes that are not related by inheritance. If the objects in a list all provide the same operations (e.g., all the objects define a certain method), then a program can process a list of those objects generically. The term polymorphism normally refers to the behavior of objects of classes related by inheritance, so we discuss polymorphic behavior in the context of class hierarchies in which all the classes in the hierarchy provide a common interface.

Consider the following example using the **Employee** base class and **HourlyWorker** derived class of Fig. 9.9. Our **Employee** base class and **HourlyWorker** derived class each define their own **__str__** methods. Calling the **__str__** method through an **Employee** reference invokes **Employee.__str__**, and calling the **__str__** method through an **HourlyWorker** reference invokes **HourlyWorker.__str__**. The base-class **__str__** method also is available to the derived class. To call the base-class **__str__** method for a derived-class object, the method must be called explicitly as follows

```
Employee.__str__( hourlyReference )
```

This specifies that the base-class **__str__** should be called explicitly, using **hourlyReference** as the object reference argument.

Through polymorphism, one method call can cause different actions to occur depending on the class of the object receiving the call. This gives the programmer tremendous expressive capability.

Software Engineering Observation 9.7

With polymorphism, the programmer can deal in generalities and let the execution-time environment concern itself with the specifics. The programmer can command a wide variety of objects to behave in manners appropriate to those objects without even knowing the types of those objects.

Software Engineering Observation 9.8

Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independently of the types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be added into such a system without modifying the base system.

Software Engineering Observation 9.9

An abstract class defines an interface for the various members of a class hierarchy. The abstract class contains methods that will be defined in the derived classes. All methods in the hierarchy can use this same interface through polymorphism.

Let us consider applications of polymorphism. A screen manager needs to display many objects of different classes, including new types that will be added to the system even after the screen manager is written. The system may need to display various shapes (i.e., base class is **`Shape`**) such as squares, circles, triangles, rectangles, points, lines and the like (each shape class is derived from the base class **`Shape`**). The screen manager uses base-class references (to **`Shape`**) to manage all the objects to be displayed. To draw any object (regardless of the level at which that object appears in the inheritance hierarchy), the screen manager simply sends a **`draw`** message to the object. Method **`draw`** has been overridden in each of the derived classes. Each object of class **`Shape`** knows how to draw itself. The screen manager does not have to worry about what type each object is or whether the object is of a type the screen manager has seen before—the screen manager simply tells each object to **`draw`** itself.

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device may operate differently from the others. Regardless of this, commands to *read* or *write* data from and to devices can have a certain uniformity. The *write* message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the *write* call itself is really no different from the *write* to any other device in the system—it simply places some number of bytes from memory onto that device. An object-oriented operating system might use an abstract base class to provide an interface appropriate for all device drivers. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly. The capabilities (i.e., the interface) offered by the device drivers are provided as methods in the abstract base class. Implementations of these methods are provided in the derived classes that correspond to the specific types of device drivers.

With polymorphic programming, a program might walk through a container, such as a list of objects from various levels of a class hierarchy. For example, a list of objects of class **`TwoDimensionalShape`** could contain objects from the derived classes **`Square`**, **`Circle`**, **`Triangle`**, **`Rectangle`**, **`Line`**, etc. Sending a message to draw each object in the list would, using polymorphism, draw the correct picture on the screen. This example of polymorphic programming highlights the benefits of a naturally polymorphic language like Python, for building large, layered systems.

9.12 Classes and Python 2.2

In versions of Python before 2.2, classes and types were two distinct programming elements. The differences between types and classes contradicts the notion that classes *are* programmer-defined types. Many Python programmers, as well as the developers of the language also disliked the limitations of this needless difference between classes and types. For example, because types are not classes, programmers cannot inherit from built-in types to take advantage of Python's high-level data manipulation capabilities provided by lists, dictionaries and other objects.

Beginning with Python 2.2, the nature and behavior of classes will change, to remove the difference between types and classes. In all future 2.x releases, a programmer can distinguish between two kinds of classes—so-called "classic" classes that behave in the same manner as the classes presented earlier in this chapter and the two preceding chapters, and "new" classes that exhibit new behavior. Python 2.2 provides type **`object`** to define new classes. Any class that directly or indirectly inherits from **`object`** exhibits all the behaviors defined for a new class, which include many advanced object-oriented features. The remainder of this section overviews some of these features in the context of live-code examples.

9.12.1 Static Methods

In Python 2.2 all classes (not only classes that inherit from **`object`**) can define *static methods*. A static method can be called by a client of the class, even if no objects of the class exist. Typically, a static method is a utility method of a class that does not require an object of the class to execute. Figure 9.10 contains an example in which we redefine class **`Employee`** to provide information about the employee's working conditions. In this example, employees work in a small office—only 10 employees can work in the office comfortably. If more than 10 employees are working in the office, it becomes too crowded and the employees are uncomfortable. Class **`Employee`** maintains a class attribute **`numberOfEmployees`** that stores the number of objects of class **`Employee`** that have been instantiated. The class also defines static method **`isCrowded`**, which determines whether the employees are working in overcrowded conditions.

Lines 4–58 contain the class **`Employee`** definition. The class defines two class attributes—**`numberOfEmployees`**, which is the number of objects of class **`Employee`** that have been created; and **`maxEmployees`**, the maximum number of employees that can work in the office comfortably.

Method **`isCrowded`** (lines 10–13) returns true if the number of existing objects of class **`Employee`** is greater than the maximum number of employees that can work in the office comfortably. The method accesses class attributes **`numberOfEmployees`** and **`maxEmployees`** through the class name (**`Employee`**). Line 16 specifies that method **`isCrowded`** is a static method for class **`Employee`**. A class designates a method as static by passing the method's name to built-in function ***`staticmethod`*** and binding a name to the value returned from the function call. Static methods differ from regular methods because, when a program calls a static method, Python does not pass the object-reference argument to the method. Therefore, a static method does not specify **`self`** as the first argument. This allows a static method to be called even if no objects of the class exist.

```python
# Fig. 9.10: EmployeeStatic.py
# Class Employee with a static method.

class Employee:
   """Employee class with static method isCrowded"""

   numberOfEmployees = 0  # number of Employees created
   maxEmployees = 10  # maximum number of comfortable employees

   def isCrowded():
      """Static method returns true if the employees are crowded"""

      return Employee.numberOfEmployees > Employee.maxEmployees

   # create static method
   isCrowded = staticmethod( isCrowded )

   def __init__( self, firstName, lastName ):
      """Employee constructor, takes first name and last name"""

      self.first = firstName
      self.last = lastName
      Employee.numberOfEmployees += 1

   def __del__( self ):
      """Employee destructor"""

      Employee.numberOfEmployees -= 1

   def __str__( self ):
      """String representation of Employee"""

      return "%s %s" % ( self.first, self.last )

# main program
def main():
   answers = [ "No", "Yes" ]  # responses to isCrowded

   employeeList = []  # list of objects of class Employee

   # call static method using class
   print "Employees are crowded?",
   print answers[ Employee.isCrowded() ]

   print "\nCreating 11 objects of class Employee..."

   # create 11 objects of class Employee
   for i in range( 11 ):
      employeeList.append( Employee( "John", "Doe" + str( i ) ) )

      # call static method using object
      print "Employees are crowded?",
      print answers[ employeeList[ i ].isCrowded() ]
```

Fig. 9.10 Static methods—class `Employee`. (Part 1 of 2.)

```
54
55      print "\nRemoving one employee..."
56      del employeeList[ 0 ]
57
58      print "Employees are crowded?", answers[ Employee.isCrowded() ]
59
60   if __name__ == "__main__":
61      main()
```

```
Employees are crowded? No

Creating 11 objects of class Employee...
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? Yes

Removing one employee...
Employees are crowded? No
```

Fig. 9.10 Static methods—class **Employee**. (Part 2 of 2.)

Method **__init__** (lines 18–23) takes two arguments that correspond to the employee's first and last name. The method also increments the value of **Employee** class attribute **numberOfEmployees**. Method **__del__** (lines 25–28) decrements the value of **Employee** class attribute **numberOfEmployees**. Method **__str__** (lines 30–33) simply returns a string that contains the employee's first and last name.

Static methods can be called either by using the class name in which the method is defined or by using the name of an object of that class. Function **main** (lines 36–58) demonstrates the ways in which a client program can call a static method. Variable **answers** (line 37) is a list that contains the possible answers (**"Yes"** or **"No"**) to the question, "Are the employees crowded?" Line 43 calls static method **isCrowded** using the class name (**Employee**). The method returns 0, because no objects of the class have been created. Lines 48–53 contain a **for** loop that creates 11 objects of class **Employee** and adds each object to list **employeeList**. For each object, the program calls static method **isCrowded** using the newest object of that class. The program prints **"Yes"** in response to the eleventh call to **isCrowded**, because the number of existing **Employees** (class attribute **numberOfEmployees**) is greater than the maximum number that can work in the office comfortably (class attribute **maxEmployees**). Line 56 deletes one of the objects from **employeeList**, which invokes that object's destructor. Line 58 calls static method **isCrowded** once more to demonstrate that the number of employees has dropped to an acceptable level.

Static methods are crucial in languages like Java which require the programmer to place all program code in a class definition. In these languages, programmers often define classes that contain only static utility methods. Clients of the class can then call the static utility methods, much in the same way the Python programs invoke functions defined in a module. In Python, static methods enable programmers to define a class interface more precisely. When a method of a class does not require an object of the class to perform its task, the programmer designates that method as static.

9.12.2 Inheriting from Built-in Types

The goal of the new class behavior is to remove the separation that existed between Python types and classes before version 2.2. The type-class unification enables programmers to define a derived class that inherits from one of Python's built-in types (e.g., integer, string and list) in the same manner that a derived class inherits from any base class. In Python 2.2, the interpreter places a reference to each type in the **__builtin__** namespace. Figure 9.11 lists common built-in type names from which a programmer-defined class can inherit. A programmer-defined class inherits from a built-in type by placing the type's name in the class's base-class list.

Figure 9.12 redefines class **SingleList**—from Section 8.12—a list that contains only unique values. The previous definition of **SingleList** (Chapter 8) defined every method for class **SingleList** that should be exposed to the client. In this example, **SingleList** inherits from base-class **list** and overrides only those methods that should provide customized behaviors in class **SingleList**. The class inherits the other methods of base-class **list**, so the programmer does not need to define the remaining **list** methods to include them as part of the new class's interface.

Class **SingleList** (Fig. 9.12) inherits from base-class **list** by placing the name **list** in the parentheses that follow the class name. Every built-in type (except **object**) inherits from **object**, so classes that inherit from built-in types (including **SingleList**)

Type name	Python data type
complex	complex number
dict	dictionary
file	file
float	floating point
int	integer
list	list
long	long integer
object	base object (*Note:* Inherit from **object** to create a "new" class.)
str	string
tuple	tuple
unicode	unicode string (*Note:* see Appendix F for information on Unicode.)

Fig. 9.11 Built-in type names in Python 2.2.

display the behaviors of “new” classes. This definition for class **SingleList** differs from our previous definition, because this definition does not maintain as an attribute an internal list of values. **SingleList** is a **list**, so all methods of the class can treat the object reference as a **list** object—an extra attribute is not necessary. Class **SingleList**’s constructor (lines 7–14) first calls the base-class constructor, to initialize the list. If the client passes an initial list value to the class’s constructor, line 14 calls **SingleList** method **merge** (discussed shortly) to add unique values from the list argument to the empty list initialized by the base-class constructor.

```
1  # Fig 9.12: NewList.py
2  # Definition for class SingleList,
3
4  class SingleList( list ):
5
6     # constructor
7     def __init__( self, initialList = None ):
8        """SingleList constructor, takes initial list value.
9        New SingleList object contains only unique values"""
10
11       list.__init__( self )
12
13       if initialList:
14          self.merge( initialList )
15
16    # utility method
17    def _raiseIfNotUnique( self, value ):
18       """Utility method to raise an exception if value
19       is in list"""
20
21       if value in self:
22          raise ValueError, \
23             "List already contains value %s" % value
24
25    # overloaded sequence operation
26    def __setitem__( self, subscript, value ):
27       """Sets value of particular index. Raises exception if list
28       already contains value"""
29
30       # terminate method on non-unique value
31       self._raiseIfNotUnique( value )
32
33       return list.__setitem__( self, subscript, value )
34
35    # overloaded mathematical operators
36    def __add__( self, other ):
37       """Overloaded addition operator, returns new SingleList"""
38
39       return SingleList( list.__add__( self, other ) )
40
```

Fig. 9.12 Inheriting from built-in type **list**—class **SingleList**. (Part 1 of 3.)

```python
   def __radd__( self, otherList ):
      """Overloaded right addition"""

      return SingleList( list.__add__( other, self ) )

   def __iadd__( self, other ):
      """Overloaded augmented assignment. Raises exception if list
      already contains any of the values in otherList"""

      for value in other:
         self.append( value )

      return self

   def __mul__( self, value ):
      """Overloaded multiplication operator. Cannot use
      multiplication on SingleLists"""

      raise ValueError, "Cannot repeat values in SingleList"

   # __rmul__ and __imul__ have same behavior as __mul__
   __rmul__ = __imul__ = __mul__

   # overridden list methods
   def insert( self, subscript, value ):
      """Inserts value at specified subscript. Raises exception if
      list already contains value"""

      # terminate method on non-unique value
      self._raiseIfNotUnique( value )

      return list.insert( self, subscript, value )

   def append( self, value ):
      """Appends value to end of list. Raises exception if list
      already contains value"""

      # terminate method on non-unique value
      self._raiseIfNotUnique( value )

      return list.append( self, value )

   def extend( self, other ):
      """Adds to list the values from another list. Raises
      exception if list already contains value"""

      for value in other:
         self.append( value )

   # new SingleList method
   def merge( self, other ):
      """Merges list with unique values from other list"""

```

Fig. 9.12 Inheriting from built-in type `list`—class `SingleList`. (Part 2 of 3.)

```
94          # add unique values from other
95          for value in other:
96
97             if value not in self:
98                list.append( self, value )
```

Fig. 9.12 Inheriting from built-in type **list**—class **SingleList**. (Part 3 of 3.)

Lines 17–23 define utility method **_raiseIfNotUnique**. This method takes as an argument a potential value to add to the list and raises an exception if the list already contains the value. All **SingleList** methods that add new elements to a list first call method **_raiseIfNotUnique**, to ensure that the client inserts only unique values in the list. Typically, a client program contains code that detects the exception, to determine whether the value was inserted successfully. [*Note*: We discuss how to detect exceptions in Chapter 12, Exception Handling.]

Method **__setitem__** (lines 26–33) executes when a client assigns a value to a particular index. The method first calls utility method **_raiseIfNotUnique** with the value to insert. If the value already is in the list, the utility method raises an exception, method **__setitem__** terminates and the value is not added to the list. If the utility method does not raise an exception, line 33 calls **__setitem__** in the base class, which either assigns the value at the specified index or, if the index is out-of-bounds, raises an exception.

```
# Fig. 9.13: fig09_13.py
# Program that uses SingleList

from NewList import SingleList

duplicates = [ 1, 2, 2, 3, 4, 3, 6, 9 ]
print "List with duplicates is:", duplicates

single = SingleList( duplicates )  # create SingleList object
print "SingleList, created from duplicates, is:", single
print "The length of the list is:", len( single )

# search for values in list
print "\nThe value 2 appears %d times in list" % single.count( 2 )
print "The value 5 appears %d times in list" % single.count( 5 )
print "The index of 9 in the list is:", single.index( 9 )

if 4 in single:
   print "The value 4 was found in list"

# add values to list
single.append( 10 )
single += [ 20 ]
single.insert( 3, "hello" )
single.extend( [ -1, -2, -3 ] )
single.merge( [ "hello", 2, 100 ] )
print "\nThe list, after adding elements is:", single
```

Fig. 9.13 Inheriting from built-in type **list**—**fig09_13.py**.

```
28
29   # remove values from list
30   popValue = single.pop()
31   print "\nRemoved", popValue, "from list:", single
32   single.append( popValue )
33   print "Added", popValue, "back to end of list:", single
34
35   # slice list
36   print "\nThe value of single[ 1:4 ] is:", single[ 1:4 ]
```

```
List with duplicates is: [1, 2, 2, 3, 4, 3, 6, 9]
SingleList, created from duplicates, is: [1, 2, 3, 4, 6, 9]
The length of the list is: 6

The value 2 appears 1 times in list
The value 5 appears 0 times in list
The index of 9 in the list is: 5
The value 4 was found in list

The list, after adding elements is: [1, 2, 3, 'hello', 4, 6, 9, 10, 20,
-1, -2, -3, 100]

Removed 100 from list: [1, 2, 3, 'hello', 4, 6, 9, 10, 20, -1, -2, -3]
Added 100 back to end of list: [1, 2, 3, 'hello', 4, 6, 9, 10, 20, -1,
-2, -3, 100]

The value of single[ 1:4 ] is: [2, 3, 'hello']
```

Fig. 9.13 Inheriting from built-in type **list**—**fig09_13.py**.

Lines 36–44 overload the **+** operator for addition when a **SingleList** appears to the left or right of the operator. Methods **__add__** and **__radd__** each return a new object of class **SingleList** that is initialized with the elements of the two arguments passed to either method. This operation has the same effect as merging two lists into one list of unique values. Lines 46–53 overload the augmented assignment **+=** symbol. The method performs its operation in-place (i.e., on the object reference itself). For each value in the right-hand operand, method **__iadd__** calls **SingleList** method **append**, which either inserts a new value at the end of the list or if the list already contains that value, raises an exception. Python expects an overloaded, augmented-assignment method to return an object of the class for which the method is defined, so line 53 returns the augmented object reference. Lines 55–62 overload the multiplication operation (i.e., list repetition) for objects of class **SingleList**. By definition, a **SingleList** cannot contain more than one occurrence of any value, so method **__mul__** raises an exception if the client attempts such an operation. Line 62 binds the names for methods **__rmul__** (right multiplication) and **__imul__** (augmented assignment multiplication) to the method defined for **__mul__**; when clients invoke these operations, the corresponding methods also raise exceptions.

Lines 65–88 define methods **insert**, **append** and **extend** for adding values to a list. Methods **insert** and **append** first invoke utility method **_raiseIfNotUnique**—to prevent the client from adding duplicate values to the list—

before invoking the base-class version of the corresponding method. Method **extend** uses method **append** to add elements from another list to the reference object.

Method **merge** (lines 91–98) provides clients the ability to merge a **SingleList** with another list that possibly contains duplicate values. Method **merge** provides the same behavior that base-class **list** provides with method **extend**. However, method **extend** in the derived class raises an exception if the client attempts to extend the **SingleList** with a list that would insert duplicate values in the **SingleList**. By providing method **merge**, we give clients a way to extend a **SingleList** without raising an exception. The method adds only unique values to the **SingleList**, by calling **list.append** for every unique value in the client-supplied list.

The driver program of Fig. 9.13 uses both **SingleList**-specific functionality and functionality inherited from base-class **list**. Lines 6–7 create and print list **duplicates**, which contains duplicate values. Line 9 creates an object of class **SingleList**, which passes **duplicates** to the constructor. The new object—**single**—of class **SingleList** contains one of each of the values from list **duplicates**. The remainder of the driver program demonstrates **SingleList**'s capabilities. Line 10 prints **single**, which implicitly invokes the object's base-class **__str__** method. Line 11 passes **single** to function **len**, which calls the object's base-class **__len__** method to determine the number of elements in the list.

Lines 14–16 call **single**'s methods **count** and **index** to determine whether certain elements exist in the list and to locate an element in the list, respectively. Line 18 uses keyword **in**, which implicitly invokes the base-class **__contains__** method, to determine whether the list contains the integer element 4. Lines 22–25 call overridden **SingleList** methods to add elements to the list. Line 22 calls method **append** to add an element to the list. Line 23 appends an element with symbol **+=**, which implicitly invokes the object's **__iadd__** method. Line 24 calls method **insert** to insert the element **"hello"** at index 3. Line 25 calls method **extend** to add elements from another list to **single**. All these methods add unique elements to the list; if one of the method calls attempted to add a duplicate value to the list, the method would raise an exception (as shown in Fig. 9.14). The call to method **merge** in line 26 merges the values in **single** with values from another list. Notice, from the output, that the effect of call in line 26 is to add only the integer element 100, because this element is the only value that **single** did not yet contain.

Lines 30–33 of Fig. 9.13 remove an element from the list, add the element back in to the list and print the results. These statements demonstrate that the client can remove a value from the list, using base-class method **pop**, and that reinserting the removed value does not raise an exception. Line 36 demonstrates that class **SingleList** inherits slicing capabilities from base-class **list**. This underscores the benefit of inheritance-based software reuse. In the previous definition of class **SingleList**, we would have had to program this capability explicitly. In this version, we simply inherit the capability from the base class.

9.12.3 __getattribute__ Method

In Chapter 8, Customizing Classes, we discussed method **__getattr__**, which executes when a client attempts to access an object attribute and that attribute name is not in the object's **__dict__**, the **__dict__** of the object's class or the **__dict__** of the class's di-

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from NewList import SingleList
>>> single = SingleList( [ 1, 2, 3 ] )
>>>
>>> single.append( 1 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 79, in append
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 1
>>>
>>> single += [ 2 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 51, in __iadd__
    self.append( value )
  File "NewList.py", line 79, in append
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 2
>>>
>>> single.insert( 0, 1 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 70, in insert
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 1
>>>
>>> single.extend( [ 3, 4 ] )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 88, in extend
    self.append( value )
  File "NewList.py", line 79, in append
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 3
```

Fig. 9.14 Class **SingleList**—inserting non-unique values.

rect and indirect base classes. Classes that inherit from base-class **object** also can define method **__*getattribute*__**, which executes for every attribute access. Figure 9.15 contains a simple example. We define class **DemostrateAccess** (lines 4–29), which inherits from base-class **object** and provides both **__getattr__** and **__getattribute__** methods. The constructor creates one attribute—**value**—and initializes it to 1.

Method **`__getattribute__`** (lines 13–19) executes every time the client attempts to access an object's attribute through the dot (**`.`**) access operator. The method prints a line indicating that the method is executing and a line that displays the name of the attribute that the client is attempting to access. Line 19 returns the result of calling base-class method **`__getattribute__`**, passing the specified attribute name. Method **`__getattribute__`** in a derived class must call the base-class version of the method to retrieve an attribute's value, because attempting to access the attribute's value through the object's **`__dict__`** would result in another call to **`__getattribute__`**.

Common Programming Error 9.5

*To ensure proper attribute access, a derived-class version of method **`__getattribute__`** should call the base-class version of the method. Attempting to return the attribute's value by accessing the object's **`__dict__`** causes infinite recursion.*

Lines 21–29 define method **`__getattr__`**, which performs the same behavior as in "classic" classes; namely, the method executes when the client attempts to access an attribute that the object's **`__dict__`** does not contain. The method displays output that indicates the method is executing and provides the name of the attribute that the client attempted to access (lines 24–26). Lines 28–29 raise an exception to preserve Python's default behavior of raising an exception when a client accesses a nonexistent attribute.

```
1  # Fig. 9.15: fig09_15.py
2  # Class that defines method __getattribute__
3
4  class DemonstrateAccess( object ):
5     """Class to demonstrate when method __getattribute__ executes"""
6
7     def __init__( self ):
8        """DemonstrateAccess constructor, initializes attribute
9        value"""
10
11       self.value = 1
12
13    def __getattribute__( self, name ):
14       """Executes for every attribute access"""
15
16       print "__getattribute__ executing..."
17       print "\tClient attempt to access attribute:", name
18
19       return object.__getattribute__( self, name )
20
21    def __getattr__( self, name ):
22       """Executes when client access attribute not in __dict__"""
23
24       print "__getattr__ executing..."
25       print "\tClient attempt to access non-existent attribute:",\
26          name
27
28       raise AttributeError, "Object has no attribute %s" \
29          % name
```

Fig. 9.15 **`__getattribute__`** method and attribute access. (Part 1 of 2).

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from fig09_15 import DemonstrateAccess
>>> access = DemonstrateAccess()
>>>
>>> access.value
__getattribute__ executing...
        Client attempt to access attribute: value
1
>>>
>>> access.novalue
__getattribute__ executing...
        Client attempt to access attribute: novalue
__getattr__ executing...
        Client attempt to access non-existent attribute: novalue
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "fig09_15.py", line 28, in __getattr__
    raise AttributeError, "Object has no attribute %s" \
AttributeError: Object has no attribute novalue
```

Fig. 9.15 **`__getattribute__`** method and attribute access. (Part 1 of 2).

The interactive session in the output box for Fig. 9.15 demonstrates when methods **`__getattribute__`** and **`__getattr__`** execute. We first create an object of class **`DemonstrateAccess`**, then access attribute **`value`**, using the dot access operator. The output indicates that method **`__getattribute__`** executes in response to the attribute access; Python displays the return value (**`1`**) in the interactive session. Next, the program accesses attribute **`novalue`**, a nonexistent attribute. Method **`__getattribute__`** executes first, because the method executes every time the client attempts to access an attribute. When the base-class version of the method determines that the object does not contain a **`novalue`** attribute, method **`__getattr__`** executes. The method raises an exception to indicate that the client has accessed a nonexistent attribute.

9.12.4 `__slots__` Class Attribute

Python's dynamism enables programmers to write applications that can change as they execute. Often, this is useful for software development purposes. For example, during the development cycle, a graphical-application programmer might create software that enables the programmer to change the application's appearance (i.e., some of the application's code) without terminating the application. This technique also is valuable for applications like Web servers that must continue executing for long periods of time, but that may need to change periodically to incorporate new features. Dynamism also has drawbacks—usually dynamic applications or applications programmed in a dynamic language exhibit poorer performance than do their non-dynamic counterparts.

One side-effect of Python's dynamic nature is that a program can add attributes to an object's namespace after the object has been created. This practice sometimes can lead to unexpected results. For example, the programmer could incorrectly type an attribute name

in an assignment statement. Rather than printing an error, such an assignment statement simply binds a new attribute name and value to the object, and the program continues executing. Python 2.2 allows new classes to define a `__slots__` attribute listing the only attributes that objects of the class are allowed to have. Figure 9.16 presents two simplified definitions of a point—classes **`PointWithoutSlots`** and **`PointWithSlots`**. A program can add attributes to objects of class **`PointWithoutSlots`**, but cannot add attributes to objects of class **`PointWithSlots`**.

```
# Fig. 9.16: Slots.py
# Simple class with slots

class PointWithoutSlots:
   """Programs can add attributes to objects of this class"""

   def __init__( self, xValue = 0.0, yValue = 0.0 ):
      """Constructor for PointWithoutSlots, initializes x- and
      y-coordinates"""

      self.x = float( xValue )
      self.y = float( yValue )

class PointWithSlots( object ):
   """Programs cannot add attributes to objects of this class"""

   # PointWithSlots objects can contain only attributes x and y
   __slots__ = [ "x", "y" ]

   def __init__( self, xValue = 0.0, yValue = 0.0 ):
      """Constructor for PointWithoutSlots, initializes x- and
      y-coordinates"""

      self.x = float( xValue )
      self.y = float( yValue )

# main program
def main():
   noSlots = PointWithoutSlots()
   slots = PointWithSlots()

   for point in [ noSlots, slots ]:
      print "\nProcessing an object of class", point.__class__

      print "The current value of point.x is:", point.x
      newValue = float( raw_input( "Enter new x coordinate: " ) )
      print "Attempting to set new x-coordinate value..."

      # Logic error: create new attribute called X, instead of
      # changing the value of attribute X
      point.X = newValue

      # output unchanged attribute x
      print "The new value of point.x is:", point.x
```

Fig. 9.16 `__slots__` attribute—specifying object attributes.

```
45
46  if __name__ == "__main__":
47     main()
```

```
Processing an object of class __main__.PointWithoutSlots
The current value of point.x is: 0.0
Enter new x coordinate: 1.0
Attempting to set new x-coordinate value...
The new value of point.x is: 0.0

Processing an object of class <class '__main__.PointWithSlots'>
The current value of point.x is: 0.0
Enter new x coordinate: 1.0
Attempting to set new x-coordinate value...
Traceback (most recent call last):
  File "Slots.py", line 47, in ?
    main()
  File "Slots.py", line 41, in main
    point.X = newValue
AttributeError: 'PointWithSlots' object has no attribute 'X'
```

Fig. 9.16 **`__slots__`** attribute—specifying object attributes.

The **`PointWithoutSlots`** definition (lines 4–12) simply defines a constructor (lines 7–12) that initializes the point's *x*- and *y*-coordinates. Class **`PointWithSlots`** (lines 14–25) inherits from base-class **`object`**, and defines an attribute **`__slots__`**—a list of attribute names that objects of the class may contain. When a new class defines the **`__slots__`** attribute, objects of the class can assign values only to attributes whose names appear in the **`__slots__`** list. If a client attempts to assign a value to an attribute whose name does not appear in **`__slots__`**, Python raises an exception.

Software Engineering Observation 9.10

*If a new class defines attribute **`__slots__`**, but the class's constructor does not initialize the attributes' values, Python assigns **`None`** to each attribute in **`__slots__`** when an object of the class is created.*

Software Engineering Observation 9.11

*A derived class inherits its base-class **`__slots__`** attribute. However, if programs should not be allowed to add attributes to objects of the derived class, the derived class must define its own **`__slots__`** attribute. The derived-class **`__slots__`** contains only the allowed derived-class attribute names, but clients still can set values for attributes specified by the derived class's direct and indirect bases classes.*

The driver program (lines 28–44) demonstrates the difference between an object of a class that defines **`__slots__`** and an object of a class that does not define **`__slots__`**. Lines 29–30 assign create objects of classes **`PointWithoutSlots`** and **`PointsWithSlots`**, respectively. The **`for`** loop in lines 32–44 iterates over each object and attempts to replace the value of the object's **`x`** attribute with a user-supplied value, obtained in line 36. Line 41 contains a logic error—the program intends to modify the value of the object's **`x`** attribute, but mistakenly creates an attribute called **`X`** and assigns the user-entered value

to the new attribute. For objects of class **PointWithoutSlots** (e.g., object **noSlots**), line 41 executes without raising an exception, and line 44 prints the unchanged value of attribute **x**. For objects of class **PointWithSlots** (e.g., **slots**), line 41 raises an exception, because the object's **__slots__** attribute does not contain the name **"X"**.

The example in Fig. 9.16 demonstrates one benefit of defining the **__slots__** attribute for new classes, namely preventing accidental attribute creation. Programs that use new classes also gain performance benefits, because Python knows in advance that programs cannot add new attributes to an object; therefore, Python can store and manipulate the objects in a more efficient manner. A disadvantage of **__slots__** is that experienced Python programmers sometimes expect the ability to add object attributes dynamically. Defining **__slots__** can inhibit programmers' abilities to create dynamic applications quickly.

9.12.5 Properties

Python's new classes can contain *properties* that describe object attributes. A program accesses an object's properties using object-attribute syntax. However, a class definition creates a property by specifying up to four components—a *get* method that executes when a program accesses the property's value, a *set* method that executes when a program sets the property's value, a *delete* method that executes when a program deletes the value (e.g., with keyword **del**) and a docstring that describes the property. The *get*, *set* and *delete* methods can perform the tasks that maintain an object's data in a consistent state. Thus, properties provide an additional way for programmers to control access to an object's data.

Figure 9.17 redefines class **Time**—the class previously used to demonstrate attribute access—to contain attributes **hour**, **minute** and **second** as properties. The constructor (lines 7–12) creates private attributes **__hour**, **__minute** and **__second**. Typically, classes that use properties define their attributes to be private, to hide the data from clients of the class. The clients of the class then access the public properties of that class, which *get* and *set* the values of the private attributes.

Method **deleteValue** (lines 20–23) raises an exception to prevent a client from deleting an attribute. We use this method to create properties that the client cannot delete. Each property (**hour**, **minute** and **second**) defines corresponding *get* and *set* methods. Each *get* method takes only the object reference as an argument and returns the property's value. Each *set* method takes two arguments—the object-reference argument and the new value for the property. Lines 25–32 define the *set* method (**setHour**) for the **hour** property. If the new value is within the appropriate range, the method assigns the new value to the property; otherwise, the method raises an exception. Method **getHour** (lines 34–37) is the **hour** property's *get* method, which simply returns the value of the corresponding private attribute (**__hour**).

```
1  # Fig. 9.17: TimeProperty.py
2  # Class Time with properties
3
4  class Time( object ):
5     """Class Time with hour, minute and second properties"""
6
```

Fig. 9.17 Properties—class **Time**. (Part 1 of 3).

```
   def __init__( self, hourValue, minuteValue, secondValue ):
      """Time constructor, takes hour, minute and second"""

      self.__hour = hourValue
      self.__minute = minuteValue
      self.__second = secondValue

   def __str__( self ):
      """String representation of an object of class Time"""

      return "%.2d:%.2d:%.2d" % \
         ( self.__hour, self.__minute, self.__second )

   def deleteValue( self ):
      """Delete method for Time properties"""

      raise TypeError, "Cannot delete attribute"

   def setHour( self, value ):
      """Set method for hour attribute"""

      if 0 <= value < 24:
         self.__hour = value
      else:
         raise ValueError, \
            "hour (%d) must be in range 0-23, inclusive" % value

   def getHour( self ):
      """Get method for hour attribute"""

      return self.__hour

   # create hour property
   hour = property( getHour, setHour, deleteValue, "hour" )

   def setMinute( self, value ):
      """Set method for minute attribute"""

      if 0 <= value < 60:
         self.__minute = value
      else:
         raise ValueError, \
            "minute (%d) must be in range 0-59, inclusive" % value

   def getMinute( self ):
      """Get method for minute attribute"""

      return self.__minute

   # create minute property
   minute = property( getMinute, setMinute, deleteValue, "minute" )

```

Fig. 9.17 Properties—class `Time`. (Part 1 of 3).

```
   def setSecond( self, value ):
      """Set method for second attribute"""

      if 0 <= value < 60:
         self.__second = value
      else:
         raise ValueError, \
            "second (%d) must be in range 0-59, inclusive" % value

   def getSecond( self ):
      """Get method for second attribute"""

      return self.__second

   # create second property
   second = property( getSecond, setSecond, deleteValue, "second" )
```

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from TimeProperty import Time
>>>
>>> time1 = Time( 5, 27, 19 )
>>> print time1
05:27:19
>>> print time1.hour, time1.minute, time1.second
5 27 19
>>>
>>> time1.hour, time1.minute, time1.second  = 16, 1, 59
>>> print time1
16:01:59
>>>
>>> time1.hour = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeProperty.py", line 31, in setHour
    raise ValueError, \
ValueError: hour (25) must be in range 0-23, inclusive
>>>
>>> time1.minute = -3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeProperty.py", line 48, in setMinute
    raise ValueError, \
ValueError: minute (-3) must be in range 0-59, inclusive
>>>
>>> time1.second = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeProperty.py", line 65, in setSecond
    raise ValueError, \
ValueError: second (99) must be in range 0-59, inclusive
```

Fig. 9.17 Properties—class `Time`. (Part 1 of 3).

Built-in function ***`property`*** (line 40) takes as arguments a *get* method, a *set* method, a *delete* method and a docstring and returns a property for the class. Line 40 creates the **`hour`** property by passing to function **`property`** methods **`getHour`**, **`setHour`** and **`deleteValue`** and the string **`"hour"`**. Clients access properties, using the dot (**`.`**) access operator. When the client uses a property as an *rvalue*, the property's *get* method executes. When the client uses the property as an *lvalue*, the property's *set* method executes. When the client deletes the property with keyword **`del`**, the property's *delete* method executes. The remainder of the class definition (lines 42–74) defines *get* and *set* methods for properties **`minute`** (created in line 57) and **`second`** (created in line 74).

Software Engineering Observation 9.12

*Function **`property`** does not require that the caller pass all four arguments. Instead, the caller can pass values for keyword arguments **`fget`**, **`fset`**, **`fdel`** and **`doc`** to specify the property's* get, set *and* delete *methods and the docstring, repsectively.*

The interactive session in Fig. 9.17 highlights the benefits of properties. A client of the class can access an object's attributes, using the dot access operator, but the class author also can ensure data integrity. Properties have added advantages over implementing methods **`__setattr__`**, **`__getattr__`** and **`__delattr__`**. For example, class authors can state explicitly the attributes for which the client may use the dot access notation. Additionally, the class author can write separate *get*, *set* and *delete* methods for each attribute, rather than using **`if`**/**`else`** logic to determine which attribute to access.

In this chapter, we discussed the mechanics of inheritance and how inheritance promotes software reuse and data abstraction. We discussed two examples of inheritance—one example of structural inheritance and one example of a class hierarchy headed by an abstract base class. We also introduced new object-oriented-programming features available in Python 2.2. We continued our discussion of data integrity by presenting properties—a feature that allows clients of the class to access data with the dot access operator and allows classes to maintain private data in a consistent state. Data hiding and data integrity are fundamental object-oriented software design principles. The topics discussed in this and the previous two chapters provide a solid foundation for programmers who want to build large, industrial-strength software systems in Python.

SUMMARY

- Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors and then overriding or embellishing these with capabilities the new classes require.
- When creating a new class, instead of writing completely new attributes and methods, the programmer can designate that the new class is to inherit the attributes and methods of a previously defined base class.
- The class that inherits from a base class is referred to as a derived class. Each derived class itself becomes a candidate to be a base class for some future derived class.
- With single inheritance, a class is derived from one base class.
- With multiple inheritance, a derived class inherits from multiple (possibly unrelated) base classes. Multiple inheritance can be complex and error prone.
- The real strength of inheritance comes from the ability to define in the derived class additions, replacements or refinements for the features inherited from the base class.

- With inheritance, every object of a derived class also may be treated as an object of that derived class's base class. However, the converse is not true—base-class objects are not objects of that base class's derived classes.
- With polymorphism, it is possible to design and implement systems that are more easily extensible. Programs can be written to process generically—as base-class objects—objects of all existing classes in a hierarchy.
- Polymorphism enables us to write programs in a general fashion to handle a wide variety of existing and yet-to-be-specified related classes.
- Object-oriented programming provides several ways of "seeing the forest through the trees"—a process called abstraction.
- "Is a" is inheritance. In an "is-a" relationship, an object of a derived-class type may also be treated as an object of the base-class type.
- "Has a" is composition. In a "has-a" relationship, an object *has* references to one or more objects of other classes as members.
- A derived class can access the attributes and methods of its base class. When a base-class member implementation is inappropriate for a derived class, that member can be overridden (i.e., replaced) in the derived class with an appropriate implementation.
- Inheritance forms tree-like hierarchical structures. A base class exists in a hierarchical relationship with its derived classes.
- Function **`issubclass`** takes two arguments that are classes and returns true if the first argument is a class that inherits from the second argument (or if the first argument is the same class as the second argument)
- Python provides a built-in function—**`isinstance`**—that determines whether an object is an object of a given class or of a subclass of that class.
- Parentheses, **`()`**, in the first line of the class definition indicates inheritance. The name of the base class (or base classes) is placed inside the parentheses.
- A direct base class of a derived class is explicitly listed inside parentheses when the derived class is defined.
- An indirect base class is not explicitly listed when the derived class is defined; rather the indirect base class is inherited from two or more levels up the class hierarchy.
- To initialize an object of a derived class, the derived-class constructor must call the base-class constructor.
- A bound method call is invoked by accessing the method name through an object. Python automatically inserts the object reference argument for bound method calls.
- An unbound method call is invoked by accessing the method through its class name then specifically passing an object.
- A class's **`__bases__`** attribute is a tuple that contains references to each of the class's base classes.
- A derived class can override a base-class method by supplying a new version of that method with the same name. When that method is mentioned by name in the derived class, the derived-class version is automatically selected.
- A base class specifies commonality. In the object-oriented design process, the designer looks for commonality and "factors it out" to form base classes. Derived classes are then customized beyond the capabilities inherited from the base class.
- A program *uses* an object if the program simply calls a method of that object through a reference.
- An object is said to have a *knows a* relationship with a second object if the first object is aware of (i.e., has a reference to) the second object. This is sometimes called an association.

- There are cases in which it is useful to define classes for which the programmer never intends to create any objects. Such classes are called abstract classes.
- The sole purpose of an abstract class is to provide an appropriate base class from which classes may inherit interface and possibly implementation. Classes from which objects can be created are called concrete classes.
- Python does not provide a way to designate an abstract class. However, the programmer can implement an abstract class by raising an exception in the class's **`__init__`** method.
- Python is inherently polymorphic because the language is dynamically typed. This means that Python determines at runtime whether an object defines a method or contains an attribute and, if so, calls the appropriate method or accesses the appropriate attribute.
- Using polymorphism, one method call can cause different actions to occur depending on the class of the object receiving the call. This gives the programmer tremendous expressive capability.
- Beginning with Python 2.2, the nature and behavior of classes will change. In all future 2.x releases, a programmer can distinguish between two kinds of classes: "classic" classes and "new" classes. In Python 3.0, all classes will behave like "new" classes.
- Python 2.2 provides type **`object`** for defining "new" classes. Any class that inherits from **`object`** exhibits the new-class behaviors.
- "New" classes can define static methods. A static method can be called by a client of the class, even if no objects of the class exist.
- A class designates a method as static by passing the method's name to built-in function **`staticmethod`** and binding a name to the value returned from the function call.
- Static methods differ from regular methods in that when a program calls a static method, Python does not pass the object reference argument to the method. Therefore, a static method does not specify **`self`** as the first argument.
- The goal of the new class behavior is to remove the dichotomy that existed between Python types and classes before version 2.2. The most practical use of this type-class unification is that programmers now can inherit from Python's built-in types.
- Classes that inherit from base-class **`object`** also can define method **`__getattribute__`**, which executes for every attribute access.
- Method **`__getattribute__`** in a derived class must call the base-class version of the method to retrieve an object's attribute; otherwise, infinite recursion occurs.
- Python 2.2 allows "new" classes to define a **`__slots__`** attribute listing the attributes that objects of the class are allowed to have.
- When a "new" class defines the **`__slots__`** attribute, objects of the class can assign values only to attributes whose names appear in the **`__slots__`** list. If a client attempts to assign a value to an attribute whose name does not appear in **`__slots__`**, Python raises an exception.
- ""New" classes can contain properties that describe object attributes. A program accesses an object's properties in the same manner as accessing the object's attributes.
- A class definition creates a property by specifying four components—a *get* method, a *set* method, a *delete* method and a docstring that describes the property. The *get*, *set* and *delete* methods can perform any tasks necessary for maintaining data in a consistent state.
- Classes that use properties most often define their attributes to be private, to hide the data from clients of the class. The clients of the class then access the public properties of that class, which *get* and *set* the values of the private attributes.
- Built-in function **`property`** takes as arguments a *get* method, a *set* method, a *delete* method and a docstring and returns a property for the class.

TERMINOLOGY

__**bases**__ attribute of a class
__**getattribute**__ method
__**slots**__ attribute of a class
"has-a" relationship
"is-a" relationship
"knows-a" relationship
"uses-a" relationship
abstract class
abstract method
abstraction
association
base class
bound method call
class library
complex type
composition
concrete class
derived class
dict type
direct base class
extensible
file type
float type
indirect base class
inherit
inheritance
int type
isinstance function
issubclass function
list type
long type
multiple inheritance
NotImplementedError exception
object base class
object type
overriding a method
polymorphism
property
property function
reusability
single inheritance
standardized reusable components
static method
staticmethod function
str type
structural inheritance
subclass
superclass
tuple type
unbound method call
unicode type

SELF-REVIEW EXERCISES

9.1 Fill in the blanks in each of the following:

a) With __________, a class is derived from several base classes.
b) In other object-oriented programming languages, like Java, the base class is called the __________ and the derived class is the __________.
c) A *has-a* relationship creates new classes by __________ of existing classes.
d) When an object has a *knows a* relationship with another object, this is an __________.
e) A base class exists in a __________ relationship with its derived classes.
f) __________ in the first line of a class definition are used to indicate inheritance.
g) An __________ is inherited from two or more levels up the class hierarchy.
h) A base class specifies __________—all classes derived from a base class inherit the capabilities of that base class.
i) __________ are classes for which the programmer never intends to create objects.
j) A __________ method does not require an object of the class to perform its operation.

9.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The derived class inherits all the attributes and methods of its base class.
b) A derived class must define a constructor that calls the base class's constructor.
c) All base classes of a derived class are explicitly listed inside parentheses when the derived class is defined.
d) To use an object of another class, a class must inherit from that class.

e) A derived class uses only the base-class methods that it overrides.
f) A derived class's constructor can invoke the base class's constructor through an unbound method call.
g) The name of the base class can be used to access the base class version of an overridden method from the derived class.
h) Placing a comma-separated list of base classes inside parentheses in a class definition indicates multiple inheritance.
i) Polymorphism enables multiple inheritance.
j) Python does not implement polymorphism.

ANSWERS TO SELF-REVIEW EXERCISES

9.1 a) multiple inheritance. b) superclass, subclass. c) composition. d) association. e) hierarchical. f) Parentheses. g) indirect base class. h) commonality. i) Abstract classes. j) static.

9.2 a) True. b) False. If a derived class does not define a constructor, Python calls the base class's constructor. c) False. Only the direct base classes of a derived class are explicitly listed. d) False. A program uses an object of another class by importing the class and creating the object or using composition to define a class that contains a reference to an object of that class. e) False. A derived class has access to all of its base class's methods. f) True. g) True. h) True. i) False. Polymorphism is the ability for objects of different classes related by inheritance to respond differently to the same message. j) False. Python is inherently polymorphic because it is dynamically typed.

EXERCISES

9.3 Study the inheritance hierarchy of Fig. 9.2. For each class, indicate some common attributes and behaviors consistent with the hierarchy. Add some other classes (e.g., **`UndergraduateStudent`**, **`GraduateStudent`**, **`Freshman`**, **`Sophomore`**, **`Junior`**, **`Senior`**, etc.) to enrich the hierarchy.

9.4 Consider the class **`Bicycle`**. Given your knowledge of some common components of bicycles, show a class hierarchy in which the class **`Bicycle`** inherits from other classes, which, in turn, inherit from yet other classes. Discuss the creation of various objects of class **`Bicycle`**. Discuss inheritance from class **`Bicycle`** for other closely related derived classes.

9.5 Many programs written with inheritance could be solved with composition instead, and vice versa. Discuss the relative merits of these approaches in the context of the **`Point`**, **`Circle`**, **`Cylinder`** class hierarchy in this chapter. Rewrite the classes in Figs. 8.6–8.8 (and the supporting programs) to use composition rather than inheritance. After you do this, reassess the relative merits of the two approaches both for the **`Point`**, **`Circle`**, **`Cylinder`** problem and for object-oriented programs in general.

9.6 Write an inheritance hierarchy for class **`Quadrilateral`**, **`Trapezoid`**, **`Parallelogram`**, **`Rectangle`** and **`Square`**. Use **`Quadrilateral`** as the base class of the hierarchy. Make the hierarchy as deep (i.e., as many levels) as possible. The data of **`Quadrilateral`** should be the *(x, y)* coordinate pairs for the four endpoints of the **`Quadrilateral`**. Write a driver program that creates and displays objects of each of these classes.

9.7 Write a function that prints a class hierarchy. The function should take one argument that is an object of a class. The function should determine the class of that object and all direct and indirect base classes of the object. [*Note:* For simplicity, assume each class in the hierarchy uses only single inheritance.] The function prints each class name on a separate line. The first line contains the top-most class in the hierarchy, and each level in the hierarchy is indented by three spaces. For example, the output for the function, when passed an object of class **`Cylinder`** from Fig. 9.8, should be:

```
Point
   Circle
      Cylinder
```

9.8 Create a class **Date** that has data members for the day, the month and the year. Modify the payroll system of Fig. 9.9 to add data members **birthDate** (an object of class **Date**) and **departmentCode** (a number) to class **Employee**. Assume this payroll is processed once per month. Then, as your program calculates the payroll for each **Employee**, add a $100.00 bonus to the person's payroll amount if this is the month in which the **Employee**'s birthday occurs.

10

Graphical User Interface Components: Part 1

Objectives

- To understand the design principles of graphical user interfaces.
- To use the **`Tkinter`** module to build graphical user interfaces.
- To create and manipulate labels, text fields, buttons, check boxes and radio buttons.
- To learn to use mouse events and keyboard events.
- To understand and use layout managers.

… the wisest prophets make sure of the event first.
Horace Walpole

Do you think I can listen all day to such stuff?
Lewis Carroll

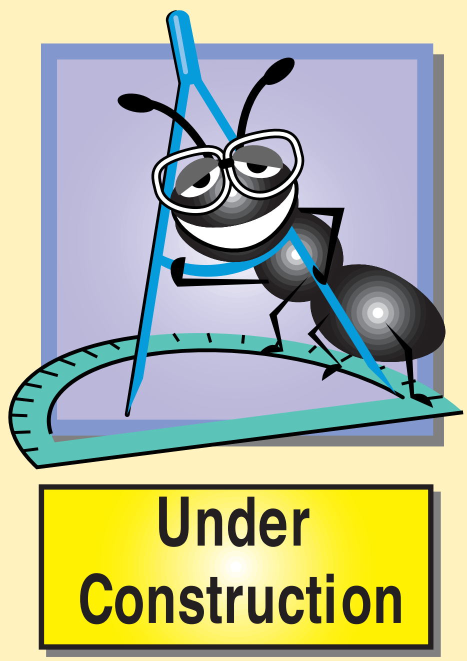
Speak the affirmative; emphasize your choice by utter ignoring of all that you reject.
Ralph Waldo Emerson

You pays your money and you takes your choice.
Punch

Guess if you can, choose if you dare.
Pierre Corneille

All hope abandon, ye who enter here!
Dante Alighieri

Exit, pursued by a bear.
William Shakespeare



Outline

10.1 Introduction

A *graphical user interface* (*GUI*) allows a user to interact with a program. A GUI (pronounced “GOO-eE”) gives a program a distinctive “look” and “feel.” Providing different programs with a consistent set of intuitive interface components provides users with a basic level of familiarity with GUI programs before they ever use them. In turn, this reduces the time users require to learn programs and increases their ability to use the programs in a productive manner.

Look-and-Feel Observation 10.1

Consistent user interfaces enable users to learn new applications faster.

GUIs are built from *GUI components* (called *widgets*—shorthand for *window gadgets*). A GUI component is an object with which a user interacts via a mouse or a keyboard. Figure 10.1 contains an example of a GUI, an Internet Explorer window with some of its GUI components labeled. There is a *menu bar* containing such *menus* as **File**, **Edit** and **View**. Below the menu bar is a set of *buttons* (e.g., **Back**, **Search**, and **History**), each of which has a defined task in Internet Explorer. Below the buttons is a *text field* in which a user can type a Web site address. To the left of the text field is a *label* (i.e., **Address**) that indicates the purpose of the text field. The menus, buttons, text fields and labels are part of the Internet Explorer GUI. These components enable a user to interact with the Internet Explorer program by just pointing with a mouse and clicking an element.

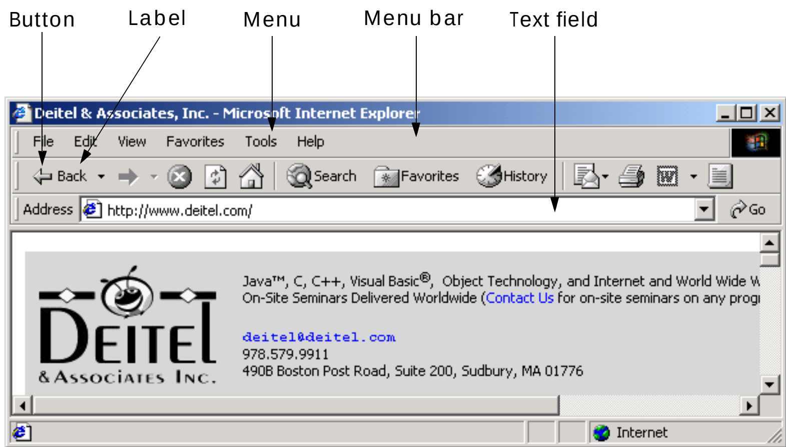


Fig. 10.1 GUI components in an Internet Explorer window.

Python programmers can construct GUIs by using the *Tool Command Language (TCL)* program and its graphic interface development tool, *Tool Kit (Tk)*. (Information about this scripting language and its components can be found at **`www.scriptics.com`**.) Figure 10.2 lists several common GUI components found in Tk. This chapter and the next discuss these and other GUI components in detail.

Component	Description
`Frame`	Serves as a container for other components.
`Label`	Displays uneditable text or icons.
`Entry`	Accepts user input from the keyboard, or displays information. A single-line input area.
`Text`	Accepts user input from the keyboard, or displays information. A multiple-line input area.
`Button`	Triggers an event when clicked.
`Checkbutton`	Selection component that is either chosen or not chosen.
`Radiobutton`	Selection component that allows the user to choose only one option.
`Menu`	Displays a list of items from which the user can select.
`Canvas`	Displays text, images, lines or shapes.
`Scale`	Allows the user to select from a range of integers using a slider.
`Listbox`	Displays a list of text options.
`Menubutton`	Displays popup or pull-down menu.
`Scrollbar`	Displays a scrollbar for canvases, text fields and lists.

Fig. 10.2 GUI components.

10.2 **`Tkinter`** Overview

The ***`Tkinter`*** module often is used to program GUIs in Python because it is Python's standard GUI package—it comes packaged with the Python program.[1] (Other GUI packages also are available for use with Python, but for this text, we use **`Tkinter`**). The **`Tkinter`** library provides an objected-oriented interface to the Tk GUI toolkit. As an object-oriented layer on top of Tk/TCL, each Tk GUI component in the **`Tkinter`** module is a class that inherits from class **`Widget`** (Fig. 10.3). All **`Widget`**-derived classes have common attributes and behaviors.

A GUI consists of a *top-level* (or, *parent*) *component* that can contain other GUI components. The components that are contained in the parent are *children* of the top-level component, and each child may contain other children. The concept of parent-child components

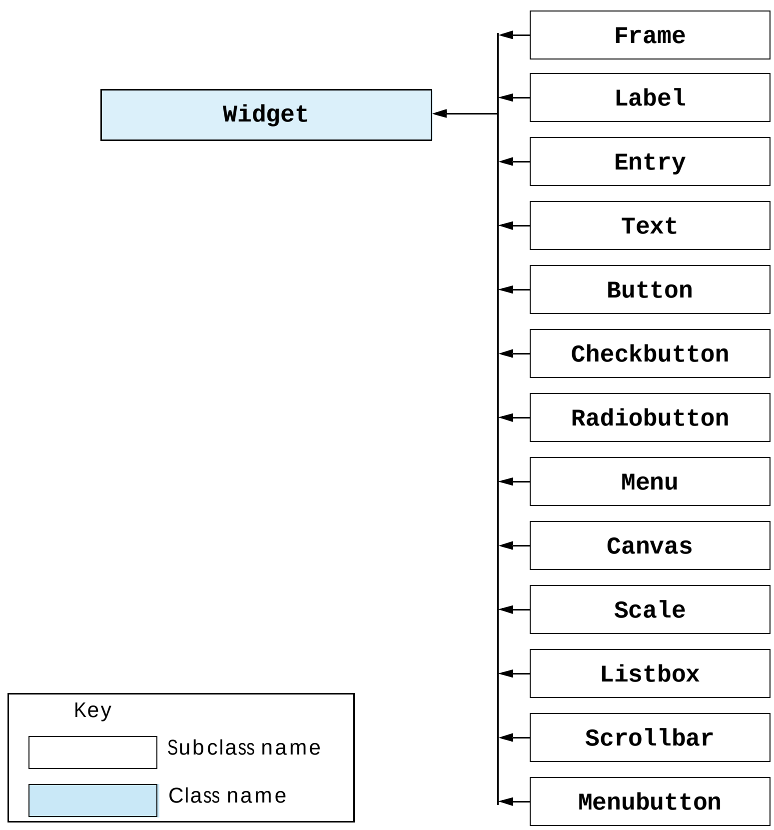


Fig. 10.3 **`Widget`** subclasses.

1. The **`Tkinter`** module is portable across many platforms. Some platforms, however, need to have Tcl/Tk and **`Tkinter`** installed. The Deitel & Associates, Inc. Web site, **`www.deitel.com`**, contains installation instructions for various platforms.

should not be confused with the relationship between a base class and a derived class. A program builds a GUI from the top-level component by creating new components and placing each new component in the parent component.

Each program in this chapter implements a GUI by inheriting from **`Widget`**'s subclass **`Frame`**. In our programs, **`Frame`** will serve as the top-level component to which children are added to extend the GUI's functionality. This inheritance enables the reuse of components in other GUI programs and promotes object-orientation.



Portability Tip 10.1

The **`Tkinter`** *module can design graphical user interfaces for Unix, Macintosh and Windows platforms.*

10.3 Simple `Tkinter` Example: `Label` Component

Labels display text or images that provide instructions or other information in graphical user interfaces. Figure 10.4 demonstrates class **`Label`**—the **`Tkinter`** class that represents a label component.

```
# Fig. 10.4: fig10_04.py
# Label demonstration.

from Tkinter import *

class LabelDemo( Frame ):
   """Demonstrate Labels"""

   def __init__( self ):
      """Create three Labels and pack them"""

      Frame.__init__( self )  # initializes Frame object

      # frame fills all available space
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Labels" )

      self.Label1 = Label( self, text = "Label with text" )

      # resize frame to accommodate Label
      self.Label1.pack()

      self.Label2 = Label( self,
         text = "Labels with text and a bitmap" )

      # insert Label against left side of frame
      self.Label2.pack( side = LEFT )

      # using default bitmap image as label
      self.Label3 = Label( self, bitmap = "warning" )
      self.Label3.pack( side = LEFT )

```

Fig. 10.4 `Label`s demonstration. (Part 1 of 2.)

```
33 def main():
34    LabelDemo().mainloop()  # starts event loop
35
36 if __name__ == "__main__":
37    main()
```



Fig. 10.4 **Label**s demonstration. (Part 2 of 2.)

Line 4 imports **Tkinter** class definitions and predefined values, or *constants*. In Chapter 4, Functions, we discussed how to import all elements from a module

```
from module import *
```

This statement allows us to write less code because specific definitions do not need to be accessed through the module's name. However, importing all definitions can cause errors. For example, if we **import *** from a module that defines a function **len**, this new definition overrides the definition for Python function **len**. If this is the case, a program cannot determine the length of a sequence. As a safeguard, only use **import *** from modules (e.g., **Tkinter**) that explicitly state that an **import *** statement may be used.

Class **LabelDemo** (lines 6–31) defines the GUI for our program. This class inherits from class **Frame** and serves as the parent container for three **Label** components. The entire GUI is constructed when a client creates a **LabelDemo** object and the class's **__init__** method (lines 9–31) executes. Line 12 calls the base class **Frame** constructor, which creates a top-level component for the entire application and initializes the **Frame**.

Once a component has been created and initialized, the component must be placed into its parent container (e.g., the top-level component created by the call to the base class constructor). Method ***pack*** (line 15) and its keyword arguments specify how and where the component should be placed in its parent. Each parent component has a certain amount of space into which child components can be placed, and each child has an original default size. When method **pack** executes, a *layout manager* determines the size and location of the child component, based on the available space in the parent container. We discuss layout managers in detail in Section 10.10.1.

The keyword argument values for method **pack** influence the size of the component. Keyword argument ***fill*** specifies how much space the component occupies, beyond its default size. Possible values for **fill** are **X** (all available horizontal space), **Y** (all available vertical space), **BOTH** (both vertical and horizontal available space) and **NONE** (the default value—occupies no additional space). Once all child components have been placed in their parent, the parent may still have available space. Keyword argument ***expand*** specifies whether a child component should occupy any extra space in its parent component (i.e., any space not yet occupied by other components). The keyword takes a value of either **YES** (expand to occupy extra space) or **NO** (do not expand to occupy extra space). The **LabelDemo** object occupies all available space provided by its parent (top-level) component because options **expand** and **fill** are set to **YES** and **BOTH**, respectively (line 15).

Look-and-Feel Observation 10.2

If no options are set, method **pack** *uses is default settings to places components in a GUI. If a programmer desires to alter the position of a component, the programmer changes the keyword arguments.*

Good Programming Practice 10.1

Before using a GUI class, read the Python online documentation to learn the methods and options of the class to understand its capabilities.

Every child component has an attribute called ***master*** that references the child's parent component. Line 16 accesses the **LabelDemo**'s parent (top-level) component and calls method ***title*** to change the title of the GUI to **Labels**, which then appears in the GUI title bar.

Line 18 creates a ***Label*** object. Each GUI component's class constructor takes a first argument that corresponds to the new object's parent. In this case, **self** is the first argument, indicating that the **Label** is a child of the **LabelDemo** component. The value of keyword argument ***text*** indicates the contents of the **Label** component. Method **pack** (line 21) inserts **Label1** into the GUI, using the default settings. By default, **Label1** occupies the top of the window.

Lines 23–24 create a second **Label** component. Line 27 calls method **pack** and passes a value for keyword argument ***side***, which describes where the new component is placed. Value **LEFT** indicates that **Label2** appears against the left side of the window. Other possible values for the **side** option are **BOTTOM**, **RIGHT** and **TOP** (the default setting). These options also determine the placing and sizing of child components when the parent container resizes. Figure 10.4 displays the resulting arrangement after the window size increases. As specified by the side option, **label1** remains at the top of the container, while **label2** and **label3** stay at the left side of the container. Section 10.10.1 discusses different settings for method pack and the effects of resizing parent containers.

A **Label**s can display an image when a programmer specifies values for the keyword argument ***bitmap***. For example, a value of **"warning"** (line 30) displays a warning *bitmap* image on **label3**. Figure 10.5 lists other values for **bitmap** that are available

Bitmap Image Name	Image	Bitmap Image Name	Image
error		**hourglass**	
gray75		**info**	
gray50		**questhead**	

Fig. 10.5 Bitmap images available. (Part 1 of 2.)

Bitmap Image Name	Image	Bitmap Image Name	Image
`gray25`		**`question`**	
`gray12`		**`warning`**	

Fig. 10.5 Bitmap images available. (Part 2 of 2.)

In addition to using existing bitmap images, programmers can create images to insert in a GUI by using keyword argument ***`image`***. Note that a hierarchy exists between **`image`**, **`bitmap`** and **`text`** keyword arguments (in that order). For example, if an image option is specified, any **`bitmap`** or **`text`** options are ignored. Similarly, if **`bitmap`** and **`text`** options both are specified, the **`text`** option is ignored. Label options follow a precedence hierarchy—the value of the option with the highest precedence appears on the GUI, and other labels are ignored. Labels with the highest precedence are **`image`**, next is **`bitmap`**, and the lowest precedence is **`text`**.

The third label component, **`Label3`**, has the **`side`** option set to **`LEFT`** (line 31). This setting left-justifies the label against **`Label2`**, not against the edge of the GUI. Section 10.10.1 offers for more information about how the **`pack`** method arranges components in a GUI.

Lines 33–37 introduce a convention common to many GUI programs. Lines 36–37 test whether the namespace is **`"__main__"`** and calls function **`main`** if the condition is true (i.e., the interpreter has been invoked on the file) and false if the file has been imported as a module. Function **`main`** executes if the program is run by itself, rather than imported as a module for use in another program.

Function **`main`** creates a **`LabelDemo`** object and calls its ***`mainloop`*** method (line 34). Method **`mainloop`** starts the **`labelDemo`** GUI. The method redraws the GUI when necessary (e.g., when the user changes the size of the GUI) and sends events to the appropriate components. [Note: We discuss events in Section 10.4.] Method **`mainloop`** terminates when the user destroys (closes) the GUI.

10.4 Event Handling Model

GUIs are *event driven*—GUI components generate *events* (actions) when users of the programs interact with the GUIs. Some common interactions include moving a mouse, clicking a mouse button, typing in a text field, selecting an item from a menu and closing a window. When a user interaction occurs, an event is sent to the program. GUI event information is stored in an object of a class **`Event`**. An event-driven program is *asynchronous*—the program does not know when events will occur.

To process a GUI event, a program must *bind* an event to a graphical component and implement an *event handler* (or *callback*). A program binds, or associates, an event with a graphical component and specifies an action to perform. An event handler is a method that is invoked in response to an associated event.

When an event occurs, the GUI component with which the user interacted determines whether an event handler has been specified for the event. If an event handler has been specified, the event handler associated with the event executes. For example, a “rollover” event occurs when the user moves the mouse over a component. A program might require that the appearance of a label changes (e.g., by changing the background color of the label) when a rollover event occurs. In this case, the programmer defines a method that changes the label’s appearance and binds the rollover event to the method. When the user moves the mouse over the label, the method executes.

10.5 `Entry` Component

`Entry` components are areas in which users can enter text or programmers can display a line of text. This section demonstrates entry components in a program. When the user types text into an **`Entry`** component and presses the *Enter* key, a ***`<Return>`*** event occurs. If an event handler is bound to that event for the **`Entry`** component, the event is processed. In our example, the **`<Return>`** event signals that the user has finished entering text in the **`Entry`**. Figure 10.6 defines class **`EntryDemo`**, which creates and manipulates four **`Entry`** text fields. When a user presses the *Enter* key in the active field, the program displays the field’s text. The program contains two **`Frame`** objects, each of which contains two **`Entry`** components.

```
# Fig. 10.6: fig10_06.py
# Entry components and event binding demonstration.

from Tkinter import *
from tkMessageBox import *

class EntryDemo( Frame ):
   """Demonstrate Entrys and Event binding"""

   def __init__( self ):
      """Create, pack and bind events to four Entrys"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Testing Entry Components" )
      self.master.geometry( "325x100" )  # width x length

      self.frame1 = Frame( self )
      self.frame1.pack( pady = 5 )

      self.text1 = Entry( self.frame1, name = "text1" )

      # bind the Entry component to event
      self.text1.bind( "<Return>", self.showContents )
      self.text1.pack( side = LEFT, padx = 5 )

      self.text2 = Entry( self.frame1, name = "text2" )

```

Fig. 10.6 **`Entry`** components and event binding demonstration. (Part 1 of 3.)

```
        # insert text into Entry component text2
        self.text2.insert( INSERT, "Enter text here" )
        self.text2.bind( "<Return>", self.showContents )
        self.text2.pack( side = LEFT, padx = 5 )

        self.frame2 = Frame( self )
        self.frame2.pack( pady = 5 )

        self.text3 = Entry( self.frame2, name = "text3" )
        self.text3.insert( INSERT, "Uneditable text field" )

        # prohibit user from altering text in Entry component text3
        self.text3.config( state = DISABLED )
        self.text3.bind( "<Return>", self.showContents )
        self.text3.pack( side = LEFT, padx = 5 )

        # text in Entry component text4 appears as *
        self.text4 = Entry( self.frame2, name = "text4",
           show = "*" )
        self.text4.insert( INSERT, "Hidden text" )
        self.text4.bind( "<Return>", self.showContents )
        self.text4.pack( side = LEFT, padx = 5 )

     def showContents( self, event ):
        """Display the contents of the Entry"""

        # acquire name of Entry component that generated event
        theName = event.widget.winfo_name()

        # acquire contents of Entry component that generated event
        theContents = event.widget.get()
        showinfo( "Message", theName + ": " + theContents )

  def main():
     EntryDemo().mainloop()

  if __name__ == "__main__":
     main()
```

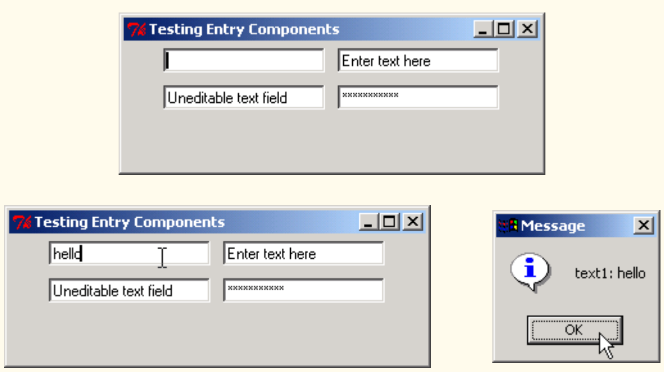


Fig. 10.6 **Entry** components and event binding demonstration. (Part 2 of 3.)

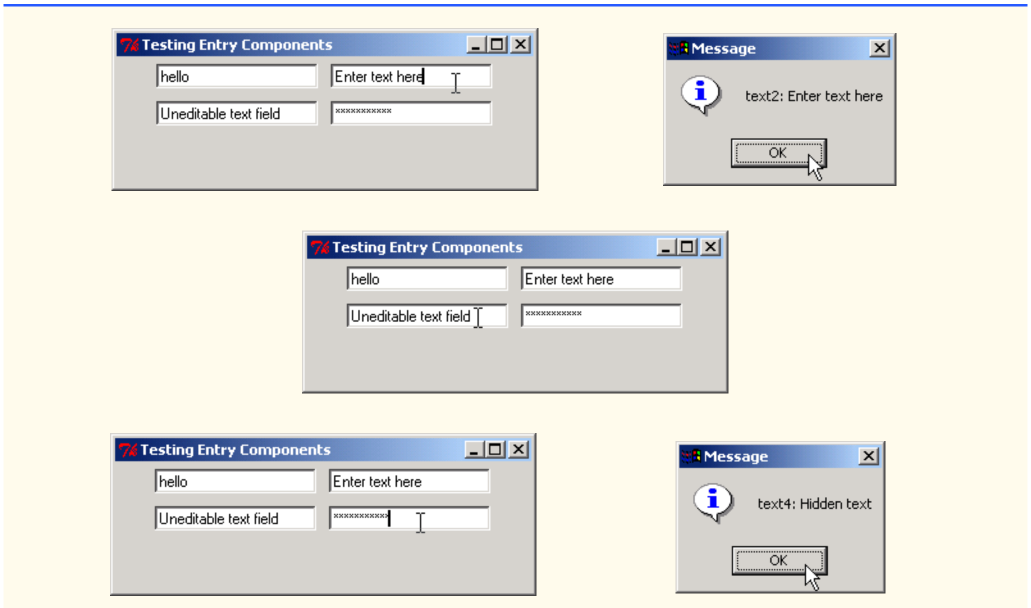


Fig. 10.6 **`Entry`** components and event binding demonstration. (Part 3 of 3.)

Line 5 imports the class definitions and constants from module ***`tkMessageBox`***. Module **`tkMessageBox`** contains functions that display *dialogs*, which present messages to users.

Class **`EntryDemo`**’s **`__init__`** method calls the base class constructor, packs the **`EntryDemo`** and titles the program (lines 13–15). Method ***`geometry`*** configures the length and width of the top-level component in pixels (line 16). Line 18 creates the first **`Frame`** component, **`frame1`**. The **`pack`** method call (line 19) introduces another option, ***`pady`***, which specifies the amount of empty vertical space between **`frame1`** and other GUI components in the parent container. Similarly, option ***`padx`***, used later in the program, specifies the amount of empty horizontal space between components.

Lines 21 create **`Entry`** component **`text1`**. Option ***`name`*** assigns a name to **`Entry`**. We assign a name so the event handler can use that name to identify the component in which an event has occurred.

Look-and-Feel Observation 10.3

If a name is not specified by the programmer, ***`Tkinter`*** *assigns each component a unique name. To obtain the full name of a component, pass the component object to function* ***`str`***.

Method ***`bind`*** (line 24) associates a **`<Return>`** event with component **`text1`**. A **`<Return>`** event occurs when the user presses the *Enter* key. Method **`bind`** takes two arguments. The first argument is the type of the event (the event format), and the second argument is the name of the method to bind to that event. In this example, method **`showContents`** executes when a **`<Return>`** event occurs in **`text1`**.

Lines 30–32 create and pack **`Entry`** component **`text2`**. Method ***`insert`*** writes text in the **`Entry`** component (line 30). Method **`insert`** takes two arguments—a position at which text is to be inserted and a string that contains the text to insert. Passing a value of

INSERT as the first argument causes the text to be inserted at the cursor's current position. Text also can be inserted at the end of an **Entry** component. For example, the call

```
insert( END, text )
```

appends **text** to the end of text already displayed in the component.

A program also can delete text from an **Entry** component with method ***delete***. The call

```
delete( start, finish )
```

removes all text in an **Entry** component in the range ***start*** to ***finish***. If **END** is the second argument, the method removes text up to the end of the text area. The first position in an **Entry** component is position 0; therefore, **delete(0**, **END)** removes all text in an **Entry** component.

Lines 34–35 creates and packs the second **Frame** component, **frame2**. The program packs the **Frame**s one below the other to create two rows into which the **Entry**s are inserted. The program inserts **Entry** components **text1** and **text2** in **frame1**, while **text3** and **text4** are packed into **frame2**.

Lines 41–43 create and pack **text3** in the same way as the first two **Entry**s. In this case, the component is bound to the **<Return>** event (line 42). In this example, we demonstrate disabling **text3** with method ***config***. Method **config** allows the user to configure a component by specifying keyword-value pairs (line 41). Specifying the value **DISABLED** for option ***state*** disables the **Entry** component, preventing the user from editing its text. As a result, **text3** cannot generate a **<Return>** event. Disabling an **Entry** can be useful to a program that wants to display text but does not want the user to edit that text.

Lines 46–50 create and pack **Entry** component **text4** in the same way as the first three **Entry**s. This component enables the user to enter confidential information. Option ***show*** specifies a character that will be displayed in the text box instead of the user-entered text (line 47). In this example, asterisks (*****) appear in place of the default text, **"Hidden text"**. Asterisks also appear in place of any text that the user types into the **Entry** component.

Method ***showContents*** (lines 52–60) is the event handler for each **<Return>** event generated in the **Entry** components. In Python, most event handlers take as a reference to an **Event** object as an argument; an **Event** object can have various attributes. The component that generated the event is obtained from the object's **widget** attribute (i.e., **event.widget**). In our program, **event.widget** refers to one of the four **Entry** components whose **<Return>** event is bound to method **showContents**.

Common Programming Error 10.1

Failure to bind an event handler to an event type for a particular GUI component results in no events being handled for that component for that event type.

Widget method ***winfo_name*** (line 56) returns the name of the component. **Entry** method ***get*** (line 59) returns the contents of the **Entry**. The event handler uses both return values to construct a message to display to the user. The **tkMessageBox** function ***showinfo*** (line 60) displays a dialog box labeled **"Message"** that contains the name and contents of the **Entry** that generated the event. The screenshots that appear at the end

of Fig. 10.6 demonstrate what happens when each **`Entry`** component receives the **`<Enter>`** event.

10.6 `Button` Component

A button is a GUI component that generates an event when it is selected. Buttons facilitate and simplify the selection of events by allowing users to select the appropriate button to execute an action, instead of manually typing commands. Buttons are created with class ***`Button`***, which inherits from class **`Widget`**. The text or image appearing on a **`Button`** component is a *button label*. A GUI can display many **`Button`**s, but, typically, each button should have a unique button label.

Look-and-Feel Observation 10.4

*Having more than one **`Button`** with the same label results in ambiguity. Provide a unique label for each button.*

Figure 10.7 creates two **`Button`**s and demonstrates that **`Button`**s, like **`Label`**s, can display both images and text.

```
# Fig. 10.7: fig10_07.py
# Button demonstration.

from Tkinter import *
from tkMessageBox import *

class PlainAndFancy( Frame ):
   """Create one plain and one fancy button"""

   def __init__( self ):
      """Create two buttons, pack them and bind events"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Buttons" )

      # create button with text
      self.plainButton = Button( self, text = "Plain Button",
         command = self.pressedPlain )
      self.plainButton.bind( "<Enter>", self.rolloverEnter )
      self.plainButton.bind( "<Leave>", self.rolloverLeave )
      self.plainButton.pack( side = LEFT, padx = 5, pady = 5 )

      # create button with image
      self.myImage = PhotoImage( file = "logotiny.gif" )
      self.fancyButton = Button( self, image = self.myImage,
         command = self.pressedFancy )
      self.fancyButton.bind( "<Enter>", self.rolloverEnter )
      self.fancyButton.bind( "<Leave>", self.rolloverLeave )
      self.fancyButton.pack( side = LEFT, padx = 5, pady = 5 )

```

Fig. 10.7 **`Button`**s demonstration. (Part 1 of 2.)

```
32     def pressedPlain( self ):
33        showinfo( "Message", "You pressed: Plain Button" )
34
35     def pressedFancy( self ):
36        showinfo( "Message", "You pressed: Fancy Button" )
37
38     def rolloverEnter( self, event ):
39        event.widget.config( relief = GROOVE )
40
41     def rolloverLeave( self, event ):
42        event.widget.config( relief = RAISED )
43
44  def main():
45     PlainAndFancy().mainloop()
46
47  if __name__ == "__main__":
48     main()
```

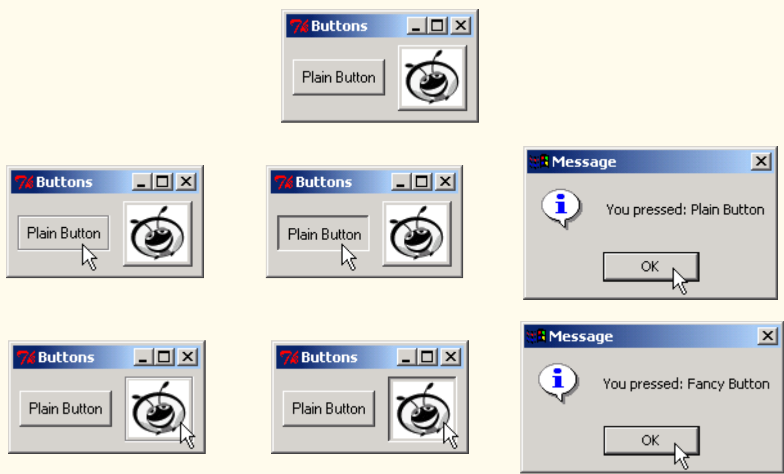


Fig. 10.7 **Button**s demonstration. (Part 2 of 2.)

Lines 18–19 create a **Button** called **plainButton**. Option **text** sets the button's label. Keyword argument ***command*** specifies the event handler that executes when a user selects the button. In our example, **plainButton**'s label is **"PlainButton"**, and its event handler is method **pressedPlain**.

Lines 20–21 bind methods ***rolloverEnter*** and ***rolloverLeave*** to **plainButton** events **<Enter>** and **<Leave>** events, respectively. The **<Enter>** event occurs when the user places the mouse cursor over the button; the **<Leave>** event occurs when the user removes the mouse cursor from the button. Section 10.8 discusses mouse events in detail.

Many **Tkinter** components, including **Button**s, can display images by specifying ***image*** arguments to their constructors or their **config** methods. The image to display must be an object of a **Tkinter** class that loads an image file. One such class is ***PhotoImage***, which supports three image formats—*Graphics Interchange Format (GIF), Joint*

Photographic Experts Group (JPEG) and *Portable Greymap Format (PGM)*. File names for each of these types typically end with **.gif**, **.jpg** (or **.jpeg**) or **.pgm** (or **.ppm**), respectively. An additional image class is class ***BitmapImage***, which supports the *Bitmap (BMP) image* format (**.bmp**). Line 25 creates a **PhotoImage** object. File **logotiny.gif** contains the image to load and store in the **PhotoImage** object. (This file resides in the same directory as the program.) The program assigns the newly created **PhotoImage** object to reference **myImage**.

Lines 26–27 create **fancyButton** with **image** attribute **myImage**. As with **Labels**, the **image** attribute takes precedence over **text** and **bitmap** attributes, and if text or bitmap are specified, they are ignored.

The event handler for **fancyButton** is **pressedFancy**. Note that methods **pressedPlain** (lines 32–33) and **pressedFancy** (lines 35–36) do not take an **Event** object as an argument. This is because **Button** callbacks do not take **Event** objects as arguments. Without an **Event** object, a callback cannot determine for which component the event occurred; therefore, it is important to specify a separate callback method for each **Button**, to ensure that the calling component can be identified. Methods **pressedPlain** and **pressedFancy** create the **"Message"** dialog boxes, which notify users of the buttons that generated the events.

Good Programming Practice 10.2

Defining a separate callback method for each ***Button*** *avoids confusion, ensures desired behavior and makes debugging a GUI easier.*

Methods **rolloverEnter** (lines 38–39) and **rolloverLeave** (lines 41–42) create a *rollover effect* for their respective events. A rollover effect changes the appearance of a component. Both methods change the *relief* of the component—how the component appears in relation to its surrounding components—for which the event occurred. Method **rolloverEnter** sets the component's ***relief*** option to **GROOVE**; method **rolloverLeave** sets **relief** to **RAISED**.

Look-and-Feel Observation 10.5

Using rollovers for ***Button****s provides users with visual feedbacks alerting them of actions that occur if the* ***Button****s are selected.*

10.7 Checkbutton and Radiobutton Components

Tkinter defines two GUI components—**Checkbutton** and **Radiobutton**—that have on/off or true/false values. Classes **Checkbutton** and **Radiobutton** are subclasses of **Widget**. Although they take the same values, class **Checkbutton** and class **Radiobutton** are used for different situations. We first discuss class **Checkbutton**.

A checkbox is a small white square that either is blank or contains a checkmark. When a checkbox is selected, a black checkmark appears in the box. There are no restrictions on how checkboxes are used—any number of boxes can be selected at a time. The text that appears alongside a checkbox is referred to as the *checkbox label*.

Figure 10.8 uses two **Checkbutton** objects to modify the font style of the text displayed in an **Entry** component. When selected, one **Checkbutton** applies a bold style, and the other applies an italic style. If both are selected, the style of the font is bold and italic. Initially, the **Checkbutton**s are not selected.

```
# Fig. 10.8: fig10_08.py
# Checkbuttons demonstration.

from Tkinter import *

class CheckFont( Frame ):
   """An area of text with Checkbutton controlled font"""

   def __init__( self ):
      """Create an Entry and two Checkbuttons"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Checkbutton Demo" )

      self.frame1 = Frame( self )
      self.frame1.pack()

      self.text = Entry( self.frame1, width = 40,
         font = "Arial 10" )
      self.text.insert( INSERT, "Watch the font style change" )
      self.text.pack( padx = 5, pady = 5 )

      self.frame2 = Frame( self )
      self.frame2.pack()

      # create boolean variable
      self.boldOn = BooleanVar()

      # create "Bold" checkbutton
      self.checkBold = Checkbutton( self.frame2, text = "Bold",
         variable = self.boldOn, command = self.changeFont )
      self.checkBold.pack( side = LEFT, padx = 5, pady = 5 )

      # create boolean variable
      self.italicOn = BooleanVar()

      # create "Italic" checkbutton
      self.checkItalic = Checkbutton( self.frame2,
         text = "Italic", variable = self.italicOn,
         command = self.changeFont )
      self.checkItalic.pack( side = LEFT, padx = 5, pady = 5 )

   def changeFont( self ):
      """Change the font based on selected Checkbuttons"""

      desiredFont = "Arial 10"

      if self.boldOn.get():
         desiredFont += " bold"

      if self.italicOn.get():
         desiredFont += " italic"
```

Fig. 10.8 `Checkbutton`s font style selection. (Part 1 of 2.)

```
54
55          self.text.config( font = desiredFont )
56
57  def main():
58     CheckFont().mainloop()
59
60  if __name__ == "__main__":
61     main()
```

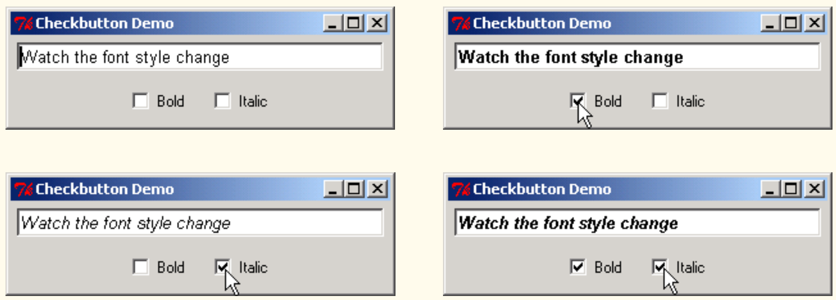


Fig. 10.8 **`Checkbutton`**s font style selection. (Part 2 of 2.)

Lines 19–20 create an **`Entry`** component named **`text`**. The inserted text, **`"Watch the font style change"`** (line 21), has font style **`"Arial 10"`**. The ***`font`*** attribute indicates the font of the **`Entry`** component. One way of representing a font is by using a string containing the font name, size and style. It is possible to specify no font style, in addition to specifying multiple font styles. The online *Introduction to Tkinter*

`www.pythonware.com/library/tkinter/introduction/x444-fonts.htm`

includes a discussion of available fonts and font styles.

`BooleanVar` objects, ***`boldOn`*** (line 28) and ***`italicOn`*** (line 36), are **`Tkinter`** integer variables that have values of either 0 or 1. The option **`variable`** requires an object of the **`Tkinter Variable`** class. **`Tkinter`** provides the ***`Variable`*** class from which **`BooleanVar`** inherits. The **`Variable`** class acts as a container for Python variables. Various **`Tkinter`** classes use **`Variable`** objects to maintain information about a particular component. For example, the **`CheckButton`** class uses a **`BooleanVar`** object to store the *state*—checked or unchecked—of the button. Our program creates and passes **`BooleanVar`** references to the **`CheckButton`** constructors, so the event handlers can determine whether the user has selected one or both of the buttons.

Lines 31–32 create a **`Checkbutton`** called **`checkBold`**. The **`text`** option indicates that the text, **`"Bold"`**, appears next to the checkbox to provide information about the purpose of the checkbox. The **`command`** attribute of a **`Checkbutton`** component is the event handler that executes when a user selects or de-selects the button. In this case, we specify method **`changeFont`** as the event handler. The component's ***`variable`*** option passes the **`BooleanVar`** object that the component uses to maintain its state information. When a user clicks the **`CheckButton`**, two things happen—its **`BooleanVar`** value changes from **`0`** to **`1`**, or **`1`** to **`0`**, and the event handler **`changeFont`** executes. Lines 38–40 create **`checkItalic`**, a **`CheckButton`** object that behaves similarly to object **`checkBold`**.

Method **`changeFont`** (lines 44–55) initializes string **`desiredFont`** to the original **`"Arial 10"`** font. Method **`get`** (of class **`BooleanVar`**) returns the variable's value. If a user selects **`checkBold`**, the program appends **`" bold"`** to **`desiredFont`** (line 50). The process repeats for **`checkItalic`**, using **`italicOn`**. Likewise, if a user selects **`checkItalic`**, the program appends the string **`" italic"`** to **`desiredFont`** (line 53). Each string begins with a space so that when the style is appended to the font, a space is included (e.g., **`"Arial 10 italic"`**). The method then calls **`config`** to change **`text`**'s font to **`desiredFont`**.

Radio buttons, created with class **`Radiobutton`**, resemble checkboxes because they have two states—*selected* and *not selected* (also called *deselected*). Unlike checkboxes, radio buttons represent a set of *mutually exclusive* options—only one radio button in a group can be selected at a time. Selecting a different radio button in the group forces all other radio buttons in the group to be deselected.

Look-and-Feel Observation 10.6

Use ***`RadioButton`****s when the user should choose only one option in a group.*

Look-and-Feel Observation 10.7

Use ***`CheckBox`****es when the user should be able to choose multiple options in a group.*

Figure 10.9 is similar to the program in Fig. 10.8 in that the user can alter the font style of an **`Entry`**'s text. However, this example permits only a single font style in the group to be selected at a time, using radio buttons.

```python
# Fig. 10.9: fig10_09.py
# Radiobuttons demonstration.

from Tkinter import *

class RadioFont( Frame ):
   """An area of text with Radiobutton controlled font"""

   def __init__( self ):
      """Create an Entry and four Radiobuttons"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Radiobutton Demo" )

      self.frame1 = Frame( self )
      self.frame1.pack()

      self.text = Entry( self.frame1, width = 40,
         font = "Arial 10" )
      self.text.insert( INSERT, "Watch the font style change" )
      self.text.pack( padx = 5, pady = 5 )

```

Fig. 10.9 **`Radiobutton`**s selecting font styles. (Part 1 of 2.)

```
24        self.frame2 = Frame( self )
25        self.frame2.pack()
26
27        fontSelections = [ "Plain", "Bold", "Italic",
28                           "Bold/Italic" ]
29        self.chosenFont = StringVar()
30
31        # initial selection
32        self.chosenFont.set( fontSelections[ 0 ] )
33
34        # create group of Radiobutton components with same variable
35        for style in fontSelections:
36           aButton = Radiobutton( self.frame2, text = style,
37              variable = self.chosenFont, value = style,
38              command = self.changeFont )
39           aButton.pack( side = LEFT, padx = 5, pady = 5 )
40
41     def changeFont( self ):
42        """Change the font based on selected Radiobutton"""
43
44        desiredFont = "Arial 10"
45
46        if self.chosenFont.get() == "Bold":
47           desiredFont += " bold"
48        elif self.chosenFont.get() == "Italic":
49           desiredFont += " italic"
50        elif self.chosenFont.get() == "Bold/Italic":
51           desiredFont += " bold italic"
52
53        self.text.config( font = desiredFont )
54
55  def main():
56     RadioFont().mainloop()
57
58  if __name__ == "__main__":
59     main()
```

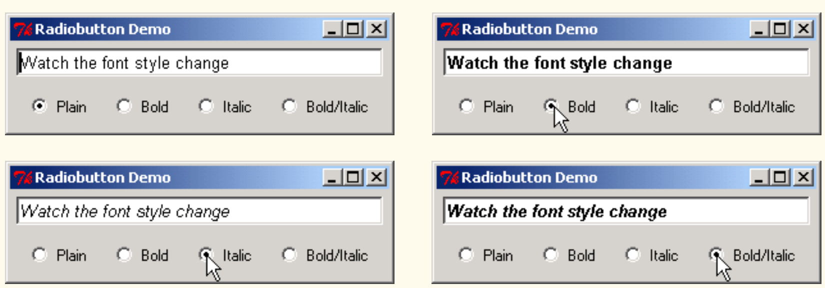


Fig. 10.9 **`Radiobutton`**s selecting font styles. (Part 2 of 2.)

Sequence ***`fontSelections`*** (lines 27–28) lists several font styles. Lines 29–32 define a ***`StringVar`*** object, ***`chosenFont`***, and **`set`**s the initial value to the default style, **`"Plain"`**. Like **`BooleanVar`**, **`StringVar`** is a subclass of **`Tkinter`** class **`Variable`**, and it acts as a container for a string variable. Unlike our **`CheckButton`**s example,

which uses a **BooleanVar** to track a button's state, a grouping of **RadioButton**s in Fig. 10.9 use a **StringVar** to store the value (i.e., name) of the selected button. Groups of RadioButtons modify the same Variable object. To define mutually exclusive groups of RadioButtons, programmers must assign one Variable object to each group. When a user selects a given radio button, the selected radio button modifies the assigned Variable object and class the appropriate event handler. Our event handler (**changeFont**) retrieves the value of the group's **StringVar** object to determine the selected button.

Lines 35–39 create and pack a **Radiobutton** component for each **style** in the **fontSelection**s list—**"Plain"**, **"Bold"**, **"Italic"** and **"Bold/Italic"**. The **for** loop assigns a **style** to each button's **text** and ***value*** options—the option that determines the button's name. Option **text** indicates the text to be displayed next to the **Radiobutton** component. Attribute **variable** associates **StringVar** object **chosenFont** with each **Radiobutton** component, and option **command** registers method **changeFont** as the event handler for each button. When the user clicks a **Radiobutton**, the string contained in the **StringVar** object is changed to contain the button's value, and method **changeFont** executes.

Method **changeFont** (lines 41–53) initializes string **desiredFont** to **"Arial 10"**. If a **Radiobutton** is selected, **changeFont** appends the desired style to **desiredFont**. Method **get** obtains the current value of **chosenFont**. In this example, **changeFont** uses an **if**/**elif** structure to emphasize that, unlike **Checkbutton**s, only one **Radiobutton** (using the same variable) may be selected at a time.

10.8 Mouse Event Handling

This section demonstrates how programs handle *mouse events*—events that occur as a result of user interaction with a mouse. Figure 10.10 summarizes several common mouse event formats and Fig. 10.11 demonstrates how a GUI program can handle them. All **Tkinter** events are described by strings following the pattern <*modifier-type-detail*>. The *type* (for instance, **Button** and **Return**) specifies the kind of event. The prefix **Double** is an example of a modifier while the specific mouse button is a detail.

Event format	Description
<ButtonPress-*n*>	Mouse button *n* has been selected while the mouse pointer is over the component. *n* may be 1 (left button), 2 (middle button) or 3 (right button). (e.g., **<ButtonPress-1>**).
<Button-*n*>, **<*n*>**	Shorthand notations for **<ButtonPress-*n*>**.
<ButtonRelease-*n*>	Mouse button *n* has been released.
<B*n*-Motion>	Mouse is moved with button *n* held down.
<*Prefix*-Button-*n*>	Mouse button *n* has been *Prefix* clicked over the component. *Prefix* may be **Double** or **Triple**.
<Enter>	Mouse pointer has entered the component.
<Leave>	Mouse pointer has exited the component.

Fig. 10.10 Mouse event formats.

```python
# Fig. 10.11: fig10_11.py
# Mouse events example.

from Tkinter import *

class MouseLocation( Frame ):
   """Demonstrate binding mouse events"""

   def __init__( self ):
      """Create a Label, pack it and bind mouse events"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Demonstrating Mouse Events" )
      self.master.geometry( "275x100" )

      self.mousePosition = StringVar() # displays mouse position
      self.mousePosition.set( "Mouse outside window" )
      self.positionLabel = Label( self,
         textvariable = self.mousePosition )
      self.positionLabel.pack( side = BOTTOM )

      # bind mouse events to window
      self.bind( "<Button-1>", self.buttonPressed )
      self.bind( "<ButtonRelease-1>", self.buttonReleased )
      self.bind( "<Enter>", self.enteredWindow )
      self.bind( "<Leave>", self.exitedWindow )
      self.bind( "<B1-Motion>", self.mouseDragged )

   def buttonPressed( self, event ):
      """Display coordinates of button press"""

      self.mousePosition.set( "Pressed at [ " + str( event.x ) +
         ", " + str( event.y ) + " ]" )

   def buttonReleased( self, event ):
      """Display coordinates of button release"""

      self.mousePosition.set( "Released at [ " + str( event.x ) +
         ", " + str( event.y ) + " ]" )

   def enteredWindow( self, event ):
      """Display message that mouse has entered window"""

      self.mousePosition.set( "Mouse in window" )

   def exitedWindow( self, event ):
      """Display message that mouse has left window"""

      self.mousePosition.set( "Mouse outside window" )

```

Fig. 10.11 Mouse events demonstration. (Part 1 of 2.)

```
52     def mouseDragged( self, event ):
53        """Display coordinates of mouse being moved"""
54
55        self.mousePosition.set( "Dragged at [ " + str( event.x ) +
56           ", " + str( event.y ) + " ]" )
57
58  def main():
59     MouseLocation().mainloop()
60
61  if __name__ == "__main__":
62     main()
```

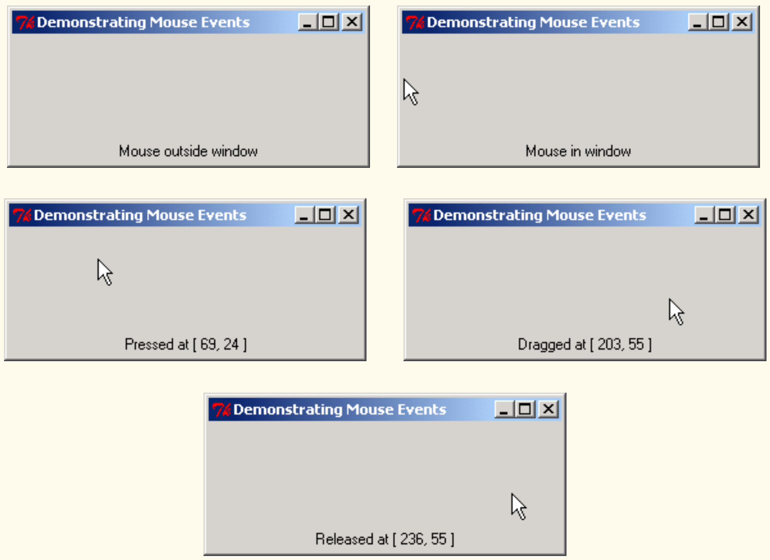


Fig. 10.11 Mouse events demonstration. (Part 2 of 2.)

Lines 17–18 create a **StringVar** object **mousePosition** and initializes its value to **"Mouse outside window"**. Lines 19–21 create and pack **Label** ***positionLabel*** with ***textvariable*** option **mousePosition**. Option **textvariable** associates the text displayed by a **Label** component with a **StringVar** object. Option **textvariable** must be associated with a **Tkinter Variable** object. (Note that in Fig. 10.4 we demonstrated the Label component's text option which is associated with a Python variable.) When the string value of the object—in this case **mousePosition**—changes, the text of the label, **positionLabel**, is updated.

Lines 24–28 bind a few common mouse events to the window. An event is generated when the left mouse button is selected or released while the mouse pointer is in the window, when the mouse pointer enters or leaves the window or when the mouse is moved with the left button pressed.

When a **<Button-1>** event or a **<ButtonRelease-1>** event is generated, method **buttonPressed** (lines 30–34) or method **buttonReleased** (lines 36–40), respectively, calls method **set** to change the value of variable **mousePosition** to inform the user of the event. A mouse event's **Event** object contains the *x*- and *y*-coordi-

nates, stored in the **x** and **y** attributes of the **Event** object, that describe where the event occurred.

When a mouse pointer enters the application area, method **enteredWindow** (lines 42–45) executes. When a mouse pointer exits the application area, method **exitedWindow** (lines 47–50) executes. As the screen captures demonstrate, each method prints an appropriate message indicating whether the mouse is over or not over the **MouseLocation** object. The methods modify the value in **StringVar** object **mousePosition** to update the **Label**'s text.

Event handler **mouseDragged** (lines 52–56) is triggered under different circumstances than event handlers **buttonPressed** and **buttonReleased**. There are two conditions which must be met before a **<B1-Motion>** event is triggered: button **B1** must be pressed and the mouse must be moving. Once these requirements are met, the **<B1-Motion>** event is fired at a rate that is defined by the operating system. In other words, on on one operating system, dragging a mouse to the right might trigger 50 **<B1-Motion>** events, while on a different operating system the rate might be much lower. For each **<B1-Motion>** event, method **mouseDragged** displays the events and the coordinates from which the event originated.

A mouse may have one, two, or three buttons. A program may need to take different actions, depending on which button the user has pressed. Figure 10.12 contains a program that demonstrates how to distinguish between different mouse buttons.

```python
# Fig. 10.12: fig10_12.py
# Mouse button differentiation.

from Tkinter import *

class MouseDetails( Frame ):
   """Demonstrate mouse events for different buttons"""

   def __init__( self ):
      """Create a Label, pack it and bind mouse events"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Mouse clicks and buttons" )
      self.master.geometry( "350x150" )

      # create mousePosition variable
      self.mousePosition = StringVar()
      positionLabel = Label( self,
         textvariable = self.mousePosition )
      self.mousePosition.set( "Mouse not clicked" )
      positionLabel.pack( side = BOTTOM )

      # bind event handler to events for each mouse button
      self.bind( "<Button-1>", self.leftClick )
      self.bind( "<Button-2>", self.centerClick )
      self.bind( "<Button-3>", self.rightClick )

```

Fig. 10.12 Mouse button differentiation. (Part 1 of 2.)

```
29     def leftClick( self, event ):
30        """Display coordinates and indicate left button clicked"""
31
32        self.showPosition( event.x, event.y )
33        self.master.title( "Clicked with left mouse button" )
34
35     def centerClick( self, event ):
36        """Display coordinates and indicate center button used"""
37
38        self.showPosition( event.x, event.y )
39        self.master.title( "Clicked with center mouse button" )
40
41     def rightClick( self, event ):
42        """Display coordinates and indicate right button clicked"""
43
44        self.showPosition( event.x, event.y )
45        self.master.title( "Clicked with right mouse button" )
46
47     def showPosition( self, x, y ):
48        """Display coordinates of button press"""
49
50        self.mousePosition.set( "Pressed at [ " + str( x ) + ", " +
51           str( y ) + " ]" )
52
53  def main():
54     MouseDetails().mainloop()
55
56  if __name__ == "__main__":
57     main()
```

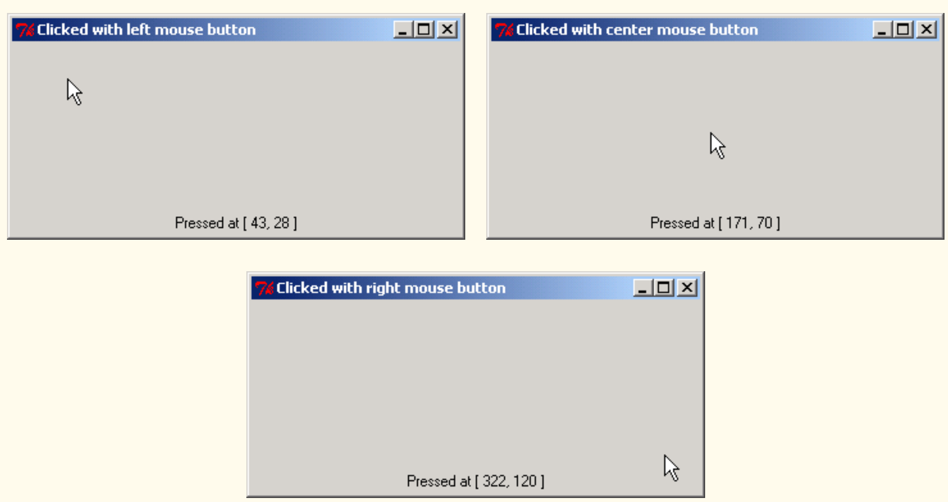


Fig. 10.12 Mouse button differentiation. (Part 2 of 2.)

Figure 10.12 is similar to Fig. 10.11 except that lines 25–27 bind methods to events for different mouse buttons by changing the number in the event format (**`<Button-`*n*`>`**). When the user presses a button while the mouse pointer is inside the window, the window's title

changes to indicate which button was pressed. Each event handler calls method **showPosition** (lines 47–51), which displays the coordinates of the mouse event.

10.9 Keyboard Event Handling

This section presents binding event handlers to *keyboard events*. These events are generated when keyboard keys are pressed and released. Figure 10.13 presents all available formats for keyboard events.

Figure 10.14 demonstrates binding methods to keyboard events. For clarity, we do not use the shorthand notations of **<KeyPress>** and **<KeyPress-*key*>** events.

Event format	Description of Event
<KeyPress>	Any key has been selected.
<KeyRelease>	Any key has been released.
<KeyPress-*key*> **<KeyRelease-*key*>**	*key* has been selected or released.
<Key>, **<Key-*key*>**	Shorthand notation for **<KeyPress>** and **<KeyPress-*key*>**.
<*key*>	Shorthand notation for **<KeyPress-*key*>**. This format works only for printable characters (excluding space and less-than sign).
<*Prefix*-*key*>	*key* has been selected while *Prefix* is held down. Possible prefixes are **Alt**, **Shift** and **Control**. Note that multiple prefixes are also possible (e.g., **<Control-Alt-*key*>**).

Fig. 10.13 Keyboard event formats.

```
# Fig. 10.14: fig10_14.py
# Binding keys to keyboard events.

from Tkinter import *

class KeyDemo( Frame ):
   """Demonstrate keystroke events"""

   def __init__( self ):
      """Create two Labels and bind keystroke events"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Demonstrating Keystroke Events" )
      self.master.geometry( "350x100" )

      self.message1 = StringVar()
      self.line1 = Label( self, textvariable = self.message1 )
```

Fig. 10.14 keyboard events demonstrated. (Part 1 of 3.)

```python
      self.message1.set( "Type any key or shift" )
      self.line1.pack()

      self.message2 = StringVar()
      self.line2 = Label( self, textvariable = self.message2 )
      self.message2.set( "" )
      self.line2.pack()

      # binding any key
      self.master.bind( "<KeyPress>", self.keyPressed )
      self.master.bind( "<KeyRelease>", self.keyReleased )

      # binding specific key
      self.master.bind( "<KeyPress-Shift_L>", self.shiftPressed )
      self.master.bind( "<KeyRelease-Shift_L>",
         self.shiftReleased )

   def keyPressed( self, event ):
      """Display the name of the pressed key"""

      self.message1.set( "Key pressed: " + event.char )
      self.message2.set( "This key is not left shift" )

   def keyReleased( self, event ):
      """Display the name of the released key"""

      self.message1.set( "Key released: " + event.char )
      self.message2.set( "This key is not left shift" )

   def shiftPressed( self, event ):
      """Display a message that left shift was pressed"""

      self.message1.set( "Shift pressed" )
      self.message2.set( "This key is left shift" )

   def shiftReleased( self, event ):
      """Display a message that left shift was released"""

      self.message1.set( "Shift released" )
      self.message2.set( "This key is left shift" )

def main():
   KeyDemo().mainloop()

if __name__ == "__main__":
   main()
```

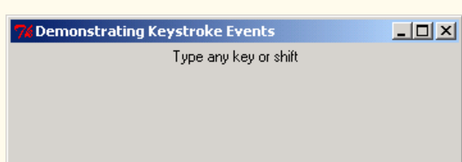


Fig. 10.14 keyboard events demonstrated. (Part 2 of 3.)

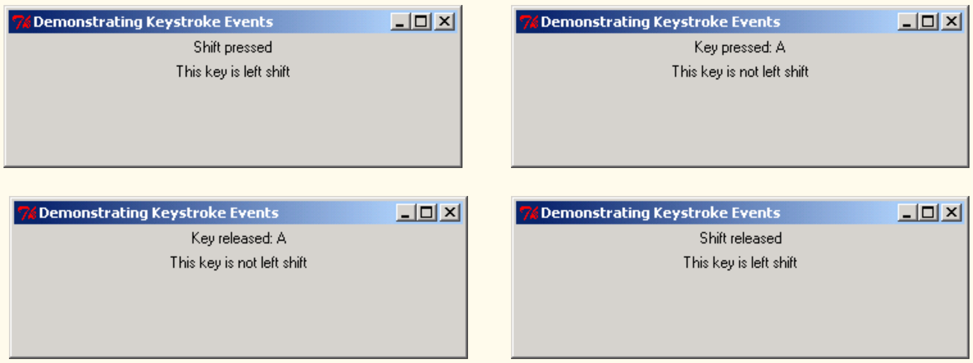


Fig. 10.14 keyboard events demonstrated. (Part 3 of 3.)

Lines 17–25 create and pack two **`Labels`**—**`line1`** and **`line2`**—that display information about the key events. Lines 28–29 bind methods **`keyPressed`** and **`keyReleased`** to **`<KeyPress>`** and **`<KeyRelease>`** events, respectively. Method **`bind`** (lines 32–34) associates the **`<KeyPress-`*n*`>`** and **`<KeyRelease-`*n*`>`** events for the left **Shift** key (***`Shift_L`***) to methods **`shiftPressed`** and **`shiftReleased`**, respectively.

Methods **`shiftPressed`** (lines 50–54) and **`shiftReleased`** (lines 56–60) display messages in the **`Label`** components when the user presses and releases the left **Shift** key, respectively. If the user selects a key other than the **Shift** key, methods **`keyPressed`** and **`keyReleased`** display messages in **`line1`** and **`line2`** indicating which key generated the event. Methods **`keyPressed`** and **`keyReleased`** obtain the name of the key with the ***`char`*** attribute of the **`Event`** object.

Portability Tip 10.2

*Not all systems can distinguish between the left and right **Shift** keys.*

10.10 Layout Managers

Layout managers arrange the placement of GUI components. Most layout managers provide basic layout capabilities that a programmer can use, rather than having to determine the exact position and size of every GUI component. Allowing layout managers to process most of the design details enables the programmer to concentrate on the basic "look and feel" of the GUI. Figure 10.15 summarizes the available layout managers.

Layout manager	Description
Pack	Places components in the order in which they were added.
Grid	Arranges components into rows and columns.
Place	Allows the programmer to specify the size and location of components and windows.

Fig. 10.15 GUI layout managers.

Good Programming Practice 10.3

Choosing the best layout manager can make programming a GUI much easier. Before programming, draw your design and select the manager that best suits it.

Common Programming Error 10.2

*Using more than one type of layout manager in the same container causes the application to freeze while **`Tkinter`** attempts to reconcile the different demands of each manager.*

10.10.1 `Pack`

All the previous GUI examples used the most basic layout manager, **`Pack`**. Unless a programmer specifies a different order, **`Pack`** places GUI components in a *container* from top to bottom in the order in which they listed in the program. A container is a GUI component into which other components may be placed. Containers are useful for managing the layout of GUI components. When the edge of the container is reached, the container expands, if possible. If the container cannot expand, the remaining components are not visible.

A programmer has several options when packing components in a container. Option **`side`** indicates the side of the container against which the component is placed. Setting **`side`** to **`TOP`** (the default value) packs components vertically. Other possible values are **`BOTTOM`**, **`LEFT`** (for horizontal placement) and **`RIGHT`**. The **`fill`** option, which can be set to **`NONE`** (default), **`X`**, **`Y`** or **`BOTH`**, allots the amount of space the component should occupy in the container. Setting **`fill`** to **`X`**, **`Y`** or **`BOTH`** ensures that a component occupies all the space the container has allocated to it in the specified direction. The **`expand`** option can be set to **`YES`** or **`NO`** (**`1`** or **`0`**). The default value is **`NO`**. If **`expand`** is set to **`YES`**, the component expands to fill any extra space in the container. The **`padx`** and **`pady`** options insert padding, or empty space, around a component. The method **`pack_forget`** removes a packed component from a container.

Good Programming Practice 10.4

Review the list of options and methods for layout managers found in the Python on-line documentation before using layout managers.

Common Programming Error 10.3

*Method **`pack`** places components in a container in the order in which they were packed; therefore, an incorrect packing order can cause undesired results. Packing components with specified values for options **`side`**, **`expand`**, **`fill`**, **`padx`** and **`pady`** can create the desired results regardless of packing order.*

Figure 10.16 creates four **`Button`**s and adds them to the application using the **`Pack`** layout manager. The example manipulates the button locations and sizes.

The **`Frame`** constructor (line 12) allows the base class to perform any initialization that it requires before we add components. Method **`title`** (line 13) displays the title in the GUI. Method **`geometry`** (line 14) sets the width and height to 300 and 150 pixels, respectively. The **`expand`** and **`fill`** options (line 15) are set to **`YES`** and **`BOTH`**, respectively, ensuring that the **`packDemo`** GUI fills the entire window. The second screen capture illustrates the GUI's appearance after it has been resized by dragging the borders with the mouse.

```python
# Fig. 10.16: fig10_16.py
# Pack layout manager demonstration.

from Tkinter import *

class PackDemo( Frame ):
   """Demonstrate some options of Pack"""

   def __init__( self ):
      """Create four Buttons with different pack options"""

      Frame.__init__( self )
      self.master.title( "Packing Demo" )
      self.master.geometry( "400x150" )
      self.pack( expand = YES, fill = BOTH )

      self.button1 = Button( self, text = "Add Button",
         command = self.addButton )

      # Button component placed against top of window
      self.button1.pack( side = TOP )

      self.button2 = Button( self,
         text = "expand = NO, fill = BOTH" )

      # Button component placed against bottom of window
      # fills all available vertical and horizontal space
      self.button2.pack( side = BOTTOM, fill = BOTH )

      self.button3 = Button( self,
         text = "expand = YES, fill = X" )

      # Button component placed against left side of window
      # fills all available horizontal space
      self.button3.pack( side = LEFT, expand = YES, fill = X )

      self.button4 = Button( self,
         text = "expand = YES, fill = Y" )

      # Button component placed against right side of window
      # fills all available vertical space
      self.button4.pack( side = RIGHT, expand = YES, fill = Y )

   def addButton( self ):
      """Create and pack a new Button"""

      Button( self, text = "New Button" ).pack( pady = 5 )

def main():
   PackDemo().mainloop()

if __name__ == "__main__":
   main()
```

Fig. 10.16 **Pack** layout manager demonstration. (Part 1 of 2.)

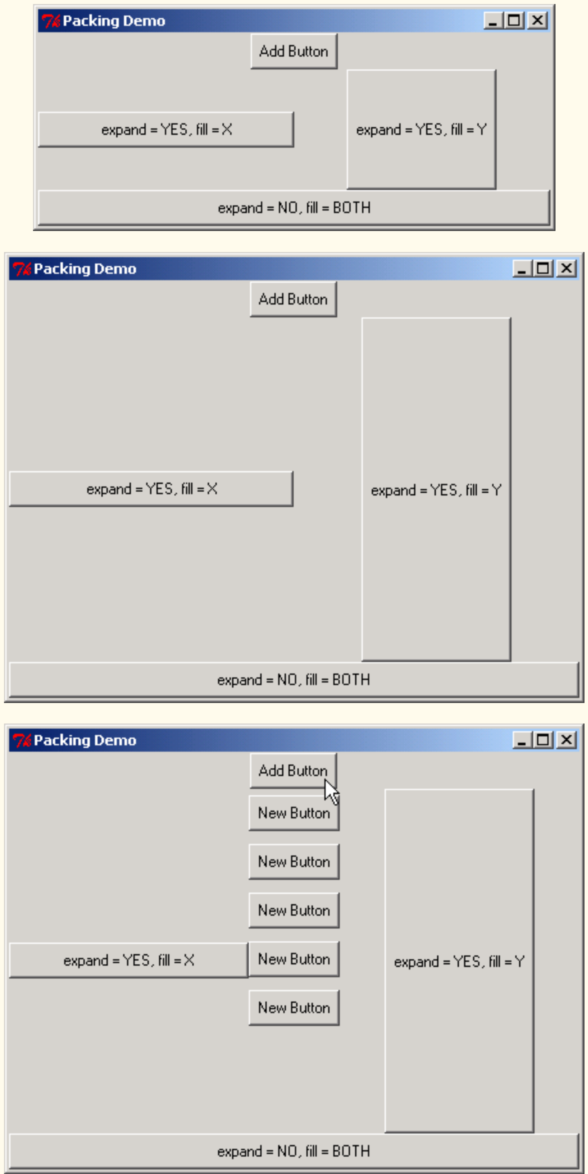


Fig. 10.16 **Pack** layout manager demonstration. (Part 2 of 2.)

Lines 17–42 create and pack four **Button**s, specifying different packing options for each **Button**. The **Pack** layout manager places each item on the top-level component in the order that they appear in the program. The specified values for options **side**, **expand** and **fill** ensure that the buttons appear as they do in the screenshots. Method **pack** (line 21) places **button1** at the top of the container as specified by option **side**. Since **fill** and **expand** are false by default, the **Button** component maintains its default size. The

next component, **button2**, (line 28) is placed at the bottom of the container. The **fill** option's value, **BOTH**, indicates that the **Button** component should occupy all space allocated to it by the container. The **expand** option is set for **button3** (line 35). Method **pack** places this component on the left side of the container. The **expand** option specifies that the button should take any available space in the container. The **X** fill option sets the button to fill all horizontal space given to it by the container. The last component, **button4**, is placed on the right side of the container. Fill option **Y** causes the button to fill all its allocated vertical space.

Only one **Button**—**button1**—specifies a callback method. When the user presses **button1**, method **addButton** (lines 44–47) creates and packs a new **Button**. The newly created **Button**s are packed vertically below **button1** and are each padded in the vertical direction by five pixels.

10.10.2 `Grid`

The ***Grid*** layout manager divides the container into a grid, so that components can be placed in rows and columns. Components are added to a grid at their specified **row** and **column** indices; every cell in the grid can contain a component. Row and column numbers begin at 0. If the **row** option is not specified, the component is placed in the first empty row and the default **column** value is 0. If the **column** option is omitted, the **column** value defaults to 0. The programmer may set the initial number of rows and columns in the grid by specifying both options in a **grid** constructor call. In addition, the rows and columns can be set with calls to methods **rowconfigure** and **columnconfigure**, respectively. Figure 10.17 demonstrates the **Grid** layout manager by placing several types of components in the GUI.

```python
# Fig. 10.17: fig10_17.py
# Grid layout manager demonstration.

from Tkinter import *

class GridDemo( Frame ):
   """Demonstrate the Grid geometry manager"""

   def __init__( self ):
      """Create and grid several components into the frame"""

      Frame.__init__( self )
      self.master.title( "Grid Demo" )

      # main frame fills entire container, expands if necessary
      self.master.rowconfigure( 0, weight = 1 )
      self.master.columnconfigure( 0, weight = 1 )
      self.grid( sticky = W+E+N+S )

      self.text1 = Text( self, width = 15, height = 5 )

```

Fig. 10.17 `Grid` layout manager demonstration. (Part 1 of 3.)

```
      # text component spans three rows and all available space
      self.text1.grid( rowspan = 3, sticky = W+E+N+S )
      self.text1.insert( INSERT, "Text1" )

      # place button component in first row, second column
      self.button1 = Button( self, text = "Button 1",
         width = 25 )
      self.button1.grid( row = 0, column = 1, columnspan = 2,
         sticky = W+E+N+S )

      # place button component in second row, second column
      self.button2 = Button( self, text = "Button 2" )
      self.button2.grid( row = 1, column = 1, sticky = W+E+N+S )

      # configure button component to fill all it allocated space
      self.button3 = Button( self, text = "Button 3" )
      self.button3.grid( row = 1, column = 2, sticky = W+E+N+S )

      # span two columns starting in second column of first row
      self.button4 = Button( self, text = "Button 4" )
      self.button4.grid( row = 2, column = 1, columnspan = 2,
         sticky = W+E+N+S )

      # place text field in fourth row to span two columns
      self.entry = Entry( self )
      self.entry.grid( row = 3, columnspan = 2,
         sticky = W+E+N+S )
      self.entry.insert( INSERT, "Entry" )

      # fill all available space in fourth row, third column
      self.text2 = Text( self, width = 2, height = 2 )
      self.text2.grid( row = 3, column = 2, sticky = W+E+N+S )
      self.text2.insert( INSERT, "Text2" )

      # make second row/column expand
      self.rowconfigure( 1, weight = 1 )
      self.columnconfigure( 1, weight = 1 )

def main():
   GridDemo().mainloop()

if __name__ == "__main__":
   main()
```

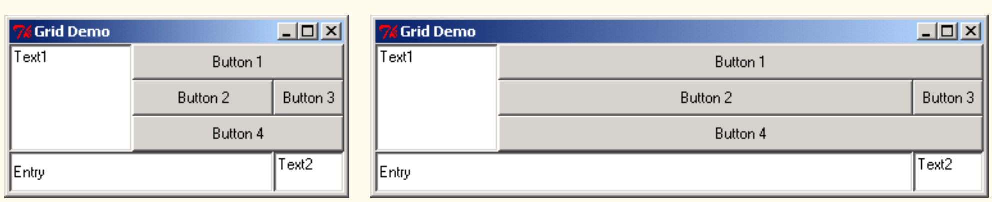


Fig. 10.17 `Grid` layout manager demonstration. (Part 2 of 3.)

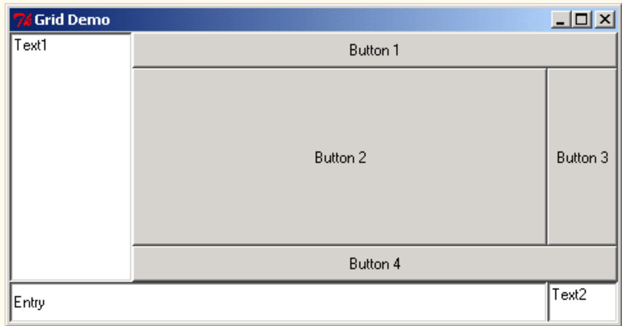


Fig. 10.17 **`Grid`** layout manager demonstration. (Part 3 of 3.)

Method **`grid`** (line 18) places the top-level component in row 0 and column 0, by default. The ***`sticky`*** option for the **`gridDemo`** object is **`W`**+**`E`**+**`N`**+**`S`**; this causes the main frame to expand to fill the entire cell. The **`sticky`** option specifies the component's alignment and whether the component stretches to fill the cell. Possible values for **`sticky`** are any combination of **`W`**, **`E`**, **`N`**, **`S`**, **`NW`**, **`NE`**, **`SW`** and **`SE`**. A **`sticky`** value of **`W`**+**`E`**, for example, is similar to setting **`fill`** to **`X`** when packing a component—the component stretches from the left (**`W`**) to the right (**`E`**) to fill the cell. Setting **`sticky`** to **`W`**+**`E`**+**`N`**+**`S`** produces results similar to those produced by the **`Pack`** layout manager's **`fill`** value of **`BOTH`**. Specifying only one value for **`sticky`** is analogous to the **`side`** option of **`Pack`**—the component aligns with the indicated cell border without being stretched. The second screenshot shows the GUI after being resized with the mouse.

The **`Grid`** manager supports several methods that control the placement of components in the container. Methods ***`rowconfigure`*** and ***`columnconfigure`*** change row and column options, respectively. For example, to ensure that row 0 stretches when the window is resized, method **`rowconfigure`** (line 16) sets the ***`weight`*** option to 1. The **`weight`** option indicates the relative weight of growth for a row or column. Weight describes the rate at which the row or column grows as the window is resized. For object, a row with a weight of three increases at three times the rate of a row whose weight is one. The default is 0—cells will not change size if a user resizes the window. Figure 10.18 describes the most common **`Grid`** methods.

`Grid` Methods	Description
`columnconfigure(` *column*, *options* **`)`**	Sets *column* options, such as **`minsize`** (minimum size), **`pad`** (add padding to largest component in the column) and **`weight`**.
`grid()`	Places a component in **`Grid`** as described by optional keyword arguments.
`grid_forget()`	Removes, but does not destroy, a component.

Fig. 10.18 **`Grid`** methods. (Part 1 of 2.)

Grid Methods	**Description**
grid_remove()	Removes a component, storing its associated options in case the component is re-inserted.
grid_info()	Returns current options as a dictionary.
grid_location(*x*, *y* **)**	Returns the grid position closest to the given pixel coordinates as a tuple (*column*, *row*).
grid_size()	Returns the grid size as a tuple (*column*, *row*).
rowconfigure(*row*, *options* **)**	Sets *row*'s options, such as **minsize** (minimum size), **pad** (add padding to largest component in the row) and **weight**.

Fig. 10.18 **Grid** methods. (Part 2 of 2.)

Line 20 introduces the ***Text*** component, which creates a multiple-line text area **text1**. Method **grid** (line 23) inserts component **text1** into the grid and introduces keyword argument ***rowspan***. The **rowspan** option sets the number of rows that a component occupies in the GUI.

Component **button1** (line 27) spans two columns, as indicated by keyword argument ***columnspan***. Option **columnspan** causes a component to stretch across a specified number of columns. Lines 27–43 create four buttons and explicitly insert each button at a certain **row** and **column**.

The **Entry** component inserted at row 3 (lines 46–49) spans two columns and fills all available space in the cell. In line 47, **columnspan** is assigned 2 and **sticky** is set to **W+E+N+S**—creating an **Entry** component that fills the first two columns of row 3. Methods **rowconfigure** and **columnconfigure** ensure that the second row and column expand when a user resizes the window (lines 57–58).

As in the **Pack** layout manager, **Grid** options **padx** and **pady** set the size of vertical and horizontal padding around a component in a cell. To place padding inside the component, use options **ipadx** and **ipady**. When a component is smaller than its cell, it is centered in the cell by default.

Common Programming Error 10.4

It is possible to specify overlapping components. The components that are packed earliest in the code are obscured by the most recently added component.

10.10.3 Place

The ***Place*** layout manager allows the user to set the position and size of a GUI component absolutely or relatively to the position and size of another component. The component being referenced is specified with the **in_** option and may be only the parent of the component being placed (default) or a descendant of its parent.

Layout manager **Place** is more complicated than the other managers. For this reason, we do not discuss the **Place** layout manager in detail, although Fig. 10.19 lists the most common **Place** methods. Figure 10.20 lists the common **place** and **place_configure** method options. For more information on layout manager **Place**, visit **www.python.org**.

Place Method	**Description**
place()	Inserts a component as specified by keyword arguments.
place_forget()	Removes, but does not destroy, a component.
place_info()	Returns current options in a dictionary.
place_configure()	Positions a component as specified by keyword arguments.

Fig. 10.19 **Place** methods.

Place Option	**Description**
x	Designates the absolute horizontal position of the component.
y	Designates the absolute vertical position of the component.
relx	Indicates the horizontal position of the component, relative to that of another component.
rely	Specifies the vertical position of the component, relative to that of another component.
width	Specifies the absolute width of the component.
height	Indicates the absolute height of the component.
relwidth	Specifies the width of the component, relative to that of another component.
relheight	Specifies the height of the component, relative to that of another component.
in_	Specifies a reference component. The newly inserted component, which must be a sibling or a child of the reference component, is placed relative to it.
anchor	Indicates which part of the component to "fix" at the given position. Possible values are **NW** (default), **N**, **NE**, **E**, **SE**, **S**, **SW**, **W** and **CENTER**.

Fig. 10.20 **Place** options.

10.11 Card Shuffling and Dealing Simulation

In this section. we use random number generation to develop a card shuffling and dealing simulation program. This program then can be used to implement programs that play specific card games.

We develop GUI class **Deck** (Fig. 10.21), which creates a deck of 52 playing cards using **Card** objects, then enables the user to deal each card by clicking on a **"Deal card"** button. Each card dealt is displayed in a **Label**. The user can shuffle the deck at any time by clicking on a **"Shuffle cards"** button.

Class **Card** (lines 7–26) contains two lists—**faces** and **suits**—that store every possible card face and suit. The constructor for the class (lines 17–21) receives a string from list **faces** and a string from list **suits**. Method **__str__** (lines 23–26) returns a string consisting of the **face** of the card, the string **" of "** and the **suit** of the card.

Class **Deck** (lines 32–95) consists of a list deck of 52 **Card** objects, an integer **currentCard** representing the most recently dealt card in the deck list and the GUI components used to manipulate the deck of cards. The constructor uses the **for** structure (lines 41–43) to fill the deck list with **Card** objects. Each **Card** is instantiated and initialized with two strings—one from the **faces** list (Strings **"Ace"** through **"King"**) and one from the **suits** list (**"Hearts"**, **"Diamonds"**, **"Clubs"** and **"Spades"**). Note that the lists are referenced as **Card.faces** and **Card.suits**, respectively, because they are class attributes of class **Card**. The calculation **i % 13** always results in a value from 0 to 12 (the thirteen subscripts of the **faces** list), and the calculation **i / 13** always results in a value from 0 to 3 (the four subscripts in the **suits** list).

```
1  # Fig. 11.21: fig11_21.py
2  # Card shuffling and dealing program
3
4  import random
5  from Tkinter import *
6
7  class Card:
8     """Class that represents one playing card"""
9
10    # class attributes faces and suits contain strings
11    # that correspond to card face and suit values
12    faces = [ "Ace", "Deuce", "Three", "Four", "Five",
13              "Six", "Seven", "Eight", "Nine", "Ten",
14              "Jack", "Queen", "King" ]
15    suits = [ "Hearts", "Diamonds", "Clubs", "Spades" ]
16
17    def __init__( self, face, suit ):
18       """Card constructor, takes face and suit as strings"""
19
20       self.face = face
21       self.suit = suit
22
23    def __str__( self ):
24       """String representation of a card"""
25
26       return "%s of %s" % ( self.face, self.suit )
27
28 class Deck( Frame ):
29    """Class to represent a GUI card deck shuffler"""
30
31    def __init__( self ):
32       """Deck constructor"""
33
34       Frame.__init__( self )
35       self.master.title( "Card Dealing Program" )
36
37       self.deck = []  # list of card objects
38       self.currentCard = 0  # index of current card
39
```

Fig. 10.21 Card-dealing program. (Part 1 of 3.)

```
      # create deck
      for i in range( 52 ):
         self.deck.append( Card( Card.faces[ i % 13 ],
            Card.suits[ i / 13 ] ) )

      # create buttons
      self.dealButton = Button( self, text = "Deal Card",
         width = 10, command = self.dealCard )
      self.dealButton.grid( row = 0, column = 0 )

      self.shuffleButton = Button( self, text = "Shuffle cards",
         width = 10, command = self.shuffle )
      self.shuffleButton.grid( row = 0, column = 1 )

      # create labels
      self.message1 = Label( self, height = 2,
         text = "Welcome to Card Dealer!" )
      self.message1.grid( row = 1, columnspan = 2 )

      self.message2 = Label( self, height = 2,
         text = "Deal card or shuffle deck" )
      self.message2.grid( row = 2, columnspan = 2 )

      self.shuffle()
      self.grid()

   def shuffle( self ):
      """Shuffle the deck"""

      self.currentCard = 0

      for i in range( len( self.deck ) ):
         j = random.randint( 0, 51 )

         # swap the cards
         self.deck[ i ], self.deck[ j ] = \
            self.deck[ j ], self.deck[ i ]

      self.message1.config( text = "DECK IS SHUFFLED" )
      self.message2.config( text = "" )
      self.dealButton.config( state = NORMAL )

   def dealCard( self ):
      """Deal one card from the deck"""

      # display the card, if it exists
      if self.currentCard < len( self.deck ):
         self.message1.config(
            text = self.deck[ self.currentCard ] )
         self.message2.config(
            text = "Card #: %d" % self.currentCard )
```

Fig. 10.21 Card-dealing program. (Part 2 of 3.)

```
91          else:
92             self.message1.config( text = "NO MORE CARDS TO DEAL" )
93             self.message2.config( text =
94                "Shuffle cards to continue" )
95             self.dealButton.config( state = DISABLED )
96
97          self.currentCard += 1  # increment card for next turn
98
99    def main():
100      Deck().mainloop()
101
102   if __name__ == "__main__":
103      main()
```

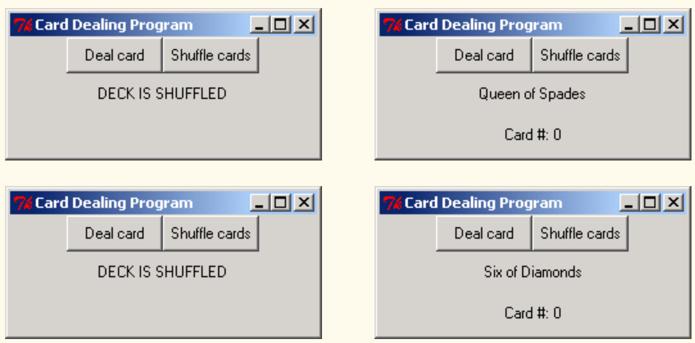


Fig. 10.21 Card-dealing program. (Part 3 of 3.)

When the user clicks the **Deal card** button, method **dealCard** (lines 82–95) gets the next card in the list. If **currentCard** is less than 52 (the length of deck), lines 87–88 display the face and suit of the card in **Label message1**. **Label message2** (lines 89–90) displays the number of the card (**currentCard**). If there are no more cards to deal (i.e., **currentCard** is greater than or equal to 52), the string **"NO MORE CARDS TO DEAL"** is displayed in **message1** and string **"Shuffle cards to continue"** is displayed in **message2**.

When the user clicks the **Shuffle cards** button, method **shuffle** (lines 66–80) shuffles the cards. The method loops through all 52 cards (list subscripts 0 to 51). For each card, a number between 0 and 51 is picked randomly. Next, the current **Card** object and the randomly selected **Card** object are swapped in the list. A total of only 52 swaps are made in a single pass of the entire list, and the list of **Card** objects is shuffled! When the shuffling is complete, **"DECK IS SHUFFLED"** is displayed in a **Label**.

10.12 Internet and World Wide Web Resources

This section presents several Internet and World Wide Web resources related to using module **Tkinter** with Python.

faqts.com/knowledge_base/index.phtml/fid/264
This **python.faqts** page contains questions and answers concerning **Tkinter** and Python interaction.

faqts.com/knowledge_base/index.phtml/fid/265
This page lists questions and answers concerning handling events.

www.pythonware.com/library/tkinter/introduction
Fredrik Lundh's *An Introduction to Tkinter* offers information about **Widget** classes and event handling.

www.python.org/topics/tkinter
This Web page provides links to documentation about **Tkinter**, additional **Widget** classes and troubleshooting tips.

www.csis.hku.hk/~kkto/doc-tkinter/tkinter/tkinter.html
Isaac K. K. To's *Building GUI Programs Using Tkinter: A Tkinter Manual* provides information about layout managers, events, the **Widget** class and subclasses.

SUMMARY

- A graphical user interface (GUI) presents a pictorial interface to a program.
- A GUI (pronounced "GOO-eE") gives a program a distinctive "look" and "feel."
- By providing different applications with a consistent set of intuitive user-interface components, GUIs allow the user to spend less time trying to remember which keystroke sequences do what and spend more time using the program in a productive manner.
- GUIs are built from GUI components (sometimes called controls or widgets—shorthand for window gadgets).
- A GUI component is an object with which a user interacts via a mouse or a keyboard.
- The **Tkinter** module is the most frequently used module for programming graphical user interfaces in Python.
- The **Tkinter** library provides an interface to the Tk (Tool Kit) GUI toolkit–the graphical interface development tool for the Tool Command Language (TCL).
- **Tkinter** implements each Tk GUI component as a class that inherits from class **Widget**.
- All **Widget**s have common attributes and behaviors.
- A GUI consists of a top-level component that may contain more GUI components. The top-level component is the parent component. The remaining components are children of the top-level component and each child of the top-level component may itself contain children (descendants of the parent component). A program builds a GUI from the top-level component by creating new components and placing each new component in its parent.
- Inheriting from class **Frame** extends the GUI's functionality. This inheritance enables the reuse of components in other GUI programs and promotes object-orientation in GUI programs.
- The **Tkinter** module, like the rest of Python, is portable across many platforms.
- Labels display text or images and usually provide instructions or other information on a graphical user interface.
- The **Frame** constructor initializes the **Frame** and creates a top-level component into which the **Frame** is placed.
- The creation of a GUI object initially does not display it on the screen. The program must specify where and how to draw the object.
- Method **pack** places components in the GUI.
- Keyword argument **fill** specifies how much available space the component occupies, beyond the component's default size. Possible values for fill are **X** (all available horizontal space), **Y** (all

available vertical space), **BOTH** (both vertical and horizontal available space) and **NONE** (the default value—do not take up available space).

- Keyword argument **expand** specifies whether a child component should take up any extra space in its parent component (i.e., any space not yet occupied once all other components have been placed).
- Each GUI component's class constructor takes a first argument that corresponds to the new object's parent.
- The value of keyword argument **text** specifies the contents of the **Label** component.
- The keyword argument **side** describes where the new component is drawn. Value **LEFT** specifies that a component is placed against the left side of the window. Other possible values for the **side** option are **BOTTOM**, **RIGHT** and **TOP**, the default setting.
- Many components display images by specifying a value for the keyword argument **bitmap**.
- Keyword argument **image** inserts a programmer-defined image. Label options have the following precedence, from highest to lowest: **image**, **bitmap** and **text**. Each **Label** component displays only one bitmap, image or text message. The value of the option with the highest precedence appears on the GUI. Any other values are ignored.
- If the interpreter is running the program, method **mainloop** method starts the GUI, redraws the GUI as needed and sends events to the appropriate components. It terminates when the user destroys (closes) the GUI.
- GUIs are event driven (i.e., they generate events when the user of the program interacts with the GUI). Some common interactions are moving the mouse, clicking a mouse button, typing in a text field, selecting an item from a menu and closing a window. When a user interaction occurs, an event is sent to the program.
- GUI event information is stored in an object of a class **Event**.
- An event-driven program is asynchronous—the program does not know when events will happen.
- To process a GUI event, the programmer must perform two key tasks—bind an event to a graphical component and implement an event handler. A program explicitly binds, or associates, an event with a graphical component and specifies an action to perform when that event occurs. Typically, the action is performed by an event handler—a method that is called in response to its associated event.
- When an event occurs, the GUI component with which the user interacted determines whether an event handler has been specified for the event. If an event handler has been specified, that event handler executes. The program can specify an event handler that executes when this event occurs.
- **Entry** components are areas in which users can enter or programmers can display a single line of text.
- When the user types data into an **Entry** component and presses the *Enter* key, a **<Return>** event occurs. If an event handler is bound to that event for the **Entry** component, the event is processed and the data in the **Entry** can be used by the program.
- Module **tkMessageBox** contains functions that display dialogs. Dialogs present messages to the user.
- Method **geometry** specifies the length and width of the top-level component in pixels.
- Option **pady** of method **pack** specifies the amount of empty vertical space between GUI components. Similarly, option **padx** specifies the amount of empty horizontal space between components.
- The **Entry** constructor's **width** argument specifies that 20 columns of text can appear in the text area on the GUI, although the **Entry** component accommodates larger inputs. The width of the text field will be the width, in pixels, of the average character in the text field's current font multiplied by 20.

- Option **name** assigns a name to the **Entry**. A program can use the name to identify the component in which an event has occurred.
- Method **bind** associates an event with a component. Method **bind** takes two arguments. The first argument is the type of the event, and the second argument is the name of the method to bind to that event.
- Method **insert** writes text in the **Entry** component. Method **insert** takes two arguments—a position at which text is to be inserted and a string that contains the text to insert.
- Passing a value of **INSERT** as the first argument to method **insert** causes the text to be inserted at the cursor's current position.
- Method call **insert(END, text)** appends **text** to the end of any text already displayed in the component.
- Method call **delete(start, finish)** removes all text in an **Entry** component in the range **start** to **finish**. Using **END** as the second argument removes text up to the end of the text area. The first position in an **Entry** component is position 0; **delete(0, END)** removes all text in an **Entry** component.
- Method **config** configures a component's options.
- Specifying the value **DISABLED** for option **state** disables the **Entry** component, preventing the user from editing its text.
- Option **show** sets the character that appears in place of the actual text.
- Most event handlers take as an argument an **Event** object, which has various attributes. The component that generated the event is obtained from the **widget** attribute of the **Event** object (i.e., **event.widget**).
- **Widget** method **winfo_name** and **Entry** method **get** acquire the name and contents of an **Entry**, respectively.
- The **tkMessageBox** function **showinfo** displays a dialog box.
- A button is a component the user clicks to trigger a specific action. A button generates an event when the user clicks the button with the mouse.
- Buttons are created with class **Button**, which inherits from class **Widget**.
- The text or image on the face of a **Button** component is called a button label.
- **Button**s (like **Label**s) can display both images and text.
- Option **text** sets the button's label.
- Keyword argument **command** specifies the event handler (or callback) that is invoked when the button is selected.
- Many **Tkinter** components, including **Button**s, can display images by specifying an **image** argument to their constructor or their **config** method.
- A specified image must be an object of a **Tkinter** class that loads an image file. One such class is **PhotoImage**, which supports three image formats—Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG) and Portable Greymap Format (PGM). File names for each of these types typically end with **.gif**, **.jpg** (or **.jpeg**) or **.pgm** (or **.ppm**), respectively.
- Class is **BitmapImage** supports the Bitmap (BMP) image format (**.bmp**).
- As with **Labels**, the **image** attribute of a **Button** component takes precedence over **text** and **bitmap** attributes.
- The **relief** option of the **Button**s is changed to **GROOVE** or **RAISED** to create rollover effects.
- **Tkinter** contains two GUI components—**Checkbutton** and **Radiobutton**—that have on/off or true/false values.

- Classes **`Checkbutton`** and **`Radiobutton`** are subclasses of **`Widget`**.
- A **`Radiobutton`** is different than a **`Checkbutton`** in that there are normally several **`Radiobutton`**s that are grouped together, and only one of the **`Radiobutton`**s in the group can be selected (true) at any time. Radio buttons are used to represent a set of mutually exclusive options (i.e., multiple options in the group cannot be selected at the same time).
- **`Entry`** fonts are specified using the **`font`** attribute. One way of representing a font is a string containing the name, size and style of the font. It is possible to specify no font style, as well as more than one font style.
- **`BooleanVar`** objects are **`Tkinter`** integer variables that have value 0 or 1.
- **`Tkinter`** provides a **`Variable`** class from which **`BooleanVar`** inherits. The **`Variable`** class acts as a container for Python variables.
- The **`CheckButton`** class uses a **`BooleanVar`** object to store the state—checked or unchecked—of the button.
- **`StringVar`**, like **`BooleanVar`**, is a subclass of **`Tkinter`** class **`Variable`**. A **`StringVar`** object acts as the interface to a string variable.
- Attribute **`variable`** associates **`chosenFont`** with each **`Radiobutton`** component.
- Keyword argument **`value`** specifies the value to assign to the associated variable when a radio button is selected.
- All **`Tkinter`** events described by strings follow the pattern **<***modifier***-***type***-***detail***>**. The type specifies the kind of event.
- An **`Event`** object for a mouse event contains information about the mouse event that occurred, including the *x*- and *y*-coordinates of the location where the event occurred. The *x*- and *y*-coordinates of the location of the mouse pointer when the event occurred are stored in the **`x`** and **`y`** attributes of the **`Event`** object.
- The name of a selected key can be obtained with the **`char`** attribute of the **`Event`** object.
- Layout managers arrange GUI components on a container for presentation purposes. Most layout managers provide basic layout capabilities that are easier to use than determining the exact position and size of every GUI component.
- Letting layout managers process most of the layout details enables the programmer to concentrate on the basic "look and feel" of the GUI.
- **`Pack`** is the most basic layout manager. GUI components are placed on a container from top to bottom in the order in which they are added to the container (unless otherwise specified). When the edge of the container is reached, the container is expanded (if possible), or the remaining components are not visible.
- A programmer can specify several options when packing components in a container.
- The **`padx`** and **`pady`** options place padding around the component.
- To remove a packed component from a container, use the component's **`pack_forget`** method.
- Method **`title`** displays the title in the GUI.
- Method **`geometry`** sets the width and height of a GUI.
- The **`Grid`** layout manager divides the container into a grid, so that components can be placed in rows and columns, where the numbering of the rows and columns starts at 0.
- Every cell in the grid can contain a component.
- Components are added to a grid at their specified **`row`** and **`column`** indices. If the **`row`** option is not specified, the component is placed in the first empty row. The default **`column`** value is 0.

- The **sticky** option specifies the component's alignment or stretches the component to fill the cell. Possible values for **sticky** are any combination of **W**, **E**, **N**, **S**, **NW**, **NE**, **SW** and **SE**.
- A **sticky** value of **W**+**E** is similar to setting **fill** to **X** when packing a component—the component stretches from the left (**W**) to the right (**E**). The component stretches horizontally to fill the cell.
- Setting **sticky** to **W**+**E**+**N**+**S** produces results similar to those produced by a **fill** value of **BOTH**.
- Specifying only one value for **sticky** is analogous to the **side** option of **Pack**. The component aligns to the specified cell border without being stretched.
- The **Grid** manager, the component into which other components have been placed, supports several methods that control the grid.
- Methods **rowconfigure** and **columnconfigure** change options of rows and columns, respectively.
- The **weight** option specifies the relative weight of growth for a row or column. The default is 0, therefore, cells will not change size if the window is resized unless the **weight** option has been changed.
- The **Text** component creates a multiple-line text area.
- The **rowspan** option sets the number of rows that a component occupies in the GUI.
- Option **columnspan** causes a component to span the specified number of columns.
- As in the **Pack** layout manager, **Grid** options **padx** and **pady** specify the size of vertical and horizontal padding around a component in a cell.
- To place padding inside the component, use options **ipadx** and **ipady**. If a component is smaller than its cell, it is centered in the cell by default.
- The **Place** layout manager allows the user to set the position and size of a GUI component in relation to the position and size of another component relatively or absolutely. The component being referenced is specified with the **in_** option and may be only the parent of the component being placed (default) or a descendant of its parent.
- The **Place** layout manager is more complicated than the other managers, so most programmers prefer to use the other, more simpler managers.

TERMINOLOGY

anchor option of layout manger **Place**
bitmap image
bitmap option of **Button** component
bitmap option of **Entry** component
BitmapImage class
<B*n***-motion>** event
BooleanVar class
BOTTOM value of option **side** of method **pack**
Button component
button label
<Button-*n***>** event
<ButtonPress-*n***>** event
<ButtonRelease-*n***>** event
callback
CENTER value of option **anchor** of layout manger **Place**
char attribute of the **Event** object
check box
Checkbutton component
children
columnconfigure method of layout manager **Grid**
column option of method **grid**
columnspan option of method **grid**
config method of class **Widget**
E value of option **anchor** of layout manger **Place**
E value of option **sticky** of method **grid**
<Enter> event
Entry component
Event class
event handler
expand option of method **pack**
fill option of method **pack**
font attribute of component **Entry**
Frame component

geometry method
get method of class **BooleanVar**
get method of **Entry** component
Graphical User Interface (GUI)
Grid layout manager
grid method
grid_forget method
grid_info method
grid_location method
grid_remove method
GUI component
height option of layout manger **Place**
image option of component **Label**
in_ option of layout manger **Place**
insert method of **Entry** component
ipadx option of method **grid**
ipady option of method **grid**
<Key> event
keyboard event
<Key-_key_**>** event
<KeyPress> event
<KeyPress-_key_**>** event
<KeyRelease> event
<KeyRelease-_key_**>** event
Label component
layout manager
<Leave> event
LEFT value of option **side** of method **pack**
Listbox component
mainloop method
menu
menu bar
Menu component
Menubutton component
mouse event
N value of option **anchor** of
layout manager **Place**
N value of option **sticky** of method **grid**
<_n_**>** event
name option of component **Entry**
NE value of option **anchor** of
layout manager **Place**
NE value of option sticky of method **grid**
NO value of option **expand** of method **pack**
NONE value of option **fill** of method **pack**
NW value of option **anchor**
layout manager **Place**
NW value of option sticky of method **grid**
Pack layout manager
pack method
pack_forget method
padx option of method **pack**
pady option of method **pack**
parent component
PhotoImage class
Place layout manager
place method
place_forget method
place_info method
<Prefix-Button-_n_**>** event
<Prefix-_key_**>** event
radio button
Radiobutton component
relheight option of layout manager **Place**
relief option of component **Button**
relwidth option of layout manager **Place**
relx option of layout manager **Place**
rely option of layout manager **Place**
<_Return_**>** event
RIGHT value of option anchor of
layout manager **Place**
row option of method **grid**
rowconfigure method of
layout manger **Grid**
rowspan option of method **grid**
S value of option **anchor** of
layout manager **Place**
S value of option **sticky** of method **grid**
Scale component
Scrollbar component
SE value of option **anchor** of
layout manger **Place**
SE value of option sticky of method **grid**
set method of class **StringVar**
show option of component **Entry**
showinfo function of module
tkMessageBox
side option of method **pack**
state option of component **Entry**
sticky option of method **grid**
StringVar class
SW value of option **anchor** of
layout manager **Place**
SW value of option sticky of method **grid**
Text component
text option of component **Label**
textvariable option of component **Label**
textvariable option of component
Radiobutton
title method

Tk (Tool Kit)
Tkinter module
tkMessageBox module
Tool Command Language (TCL)
TOP value of option **side** of method **pack**
top-level component
value option of component **Checkbutton**
value option of component **Radiobutton**
variable option of component **Checkbutton**
variable option of component **Radiobutton**
W value of option anchor of layout manager **Place**
W value of option sticky of method **grid**
weight option of layout manager **Grid**
Widget class
width option of layout manager **Place**
winfo_name method of class **Widget**
X option of layout manager **Place**
x value of option **fill** of method **pack**
Y option of layout manager **Place**
y value of option **fill** of method **pack**
YES value of option **expand** of method **pack**

SELF-REVIEW EXERCISES

10.1 Fill in the blanks in each of the following:

a) A __________ presents a pictorial user interface to a program.
b) Labels are defined with class __________— a subclass of __________.
c) __________ are single-line areas in which text can be displayed.
d) Method __________ displays text in an **Entry**.
e) Method __________ displays a message dialog.
f) A __________ is a container for other components.
g) Use method __________ of class __________ to acquire the name of an **Entry**.
h) __________ arrange GUI components on a container for presentation purposes.
i) A __________ is a component that the user clicks to trigger an action.
j) The __________ places components in the specified row and column.

10.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) All Tkinter classes inherit from **Frame**.
b) A **Label** displays only text.
c) The **Entry** component creates multiple-line text areas.
d) When the user types data into an **Entry** and presses the Enter key, an **<Enter>** event occurs.
e) **Tkinter Button** components display images using method **img**.
f) Class **PhotoImage** supports GIF, JPEG and PGM images.
g) Only one **Radiobutton** can be selected at a time.
h) **Boolean** objects are **Tkinter** integer variables that can have a value of 0 or 1.
i) Event format **<Left>** handles the event in which a mouse pointer has exited the component.
j) Layout managers arrange the placement of GUI components.

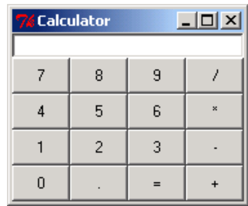
ANSWERS TO SELF-REVIEW EXERCISES

10.1 a) graphical user interface (GUI). b) **Label**, **Widget**. c) **Entry**s. d) **insert**. e) **showinfo**. f) **Frame**. g) **winfo_name**, **Widget**. h) Layout managers. i) button. j) **Grid** layout manager.

10.2 a) False. All Tkinter classes inherit from **Widget**. b) False. A **Label** can display text or an image. c) False. The **Entry** component creates single-line text areas. A **Text** widget creates multiple-line text areas. d) False. When the user types data into an **Entry** and presses the Enter key, a **<Return>** event occurs. e) False. **Tkinter Button** components display images using method **image**. f) True. g) True. h) False. **BooleanVar** objects are **Tkinter** integer variables that can have a value of 0 or 1. i) False. Event format **<Leave>** handles the event in which a mouse pointer has left the component. j) True.

EXERCISES

10.3 Create the following GUI using the **`Grid`** layout manager. You do not have to provide any functionality.



10.4 Write a temperature conversion program that converts Fahrenheit to Celsius. Use the **`Pack`** layout manager. The Fahrenheit temperature should be entered from the keyboard via an **`Entry`** component. A **`tkMessageBox`** should display the converted temperature. Use the following formula for the conversion:

```
Celsius = 5 /  9 * (Fahrenheit - 32)
```

10.5 Enhance the temperature conversion program of Exercise 10.4 by adding the Kelvin temperature scale. The program should also allow the user to make conversions between any two scales. Use the following formula for the conversion between Kelvin and Celsius (in addition to the formula in Exercise 10.4):

```
Kelvin = Celsius + 273
```

10.6 Add functionality—addition, subtraction, multiplication and division—to the calculator created in Exercise 10.3. Use the built-in Python function **`eval`** to evaluates strings. For instance, **`eval( "34+24" )`** returns the integer 58.

10.7 Write a program that allows the user to practice typing. When the user clicks a button, the program generates and displays a random sequence of letters in an **`Entry`** component. The user repeats the sequence in another **`Entry`** component. When the user enters an incorrect letter, the program displays an error message until the user types the correct letter. Use keyboard events.

10.8 Create a GUI for a matching game. Initially, buttons should cover pairs of images. When the user clicks a button, the image displays. If the user finds a matching pair, disable the buttons and display their images. If the user's choices do not match, hide the images.

11

Graphical User Interface Components: Part 2

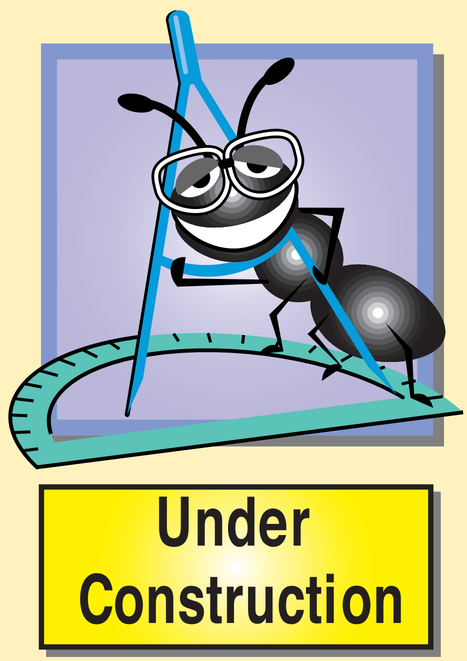
Objectives

- To create a scrolled list of items from which a user can make a selection.
- To create scrolled text areas.
- To create menus and popup menus.
- To create and manipulate canvases and scales.

I claim not to have controlled events, but confess plainly that events have controlled me.
Abraham Lincoln

A good symbol is the best argument, and is a missionary to persuade thousands.
Ralph Waldo Emerson

Capture its reality in paint!
Paul Cézanne



Outline

11.1 Introduction

In this chapter, we continue our study of GUIs. We discuss more advanced components and lay the groundwork for building complex GUIs.

We discuss *Python megawidgets (**`Pmw`**)*—a toolkit that provides high-level GUI components developed from smaller components provided by the **`Tkinter`** module. For example, a **`Pmw`** ***`ScrolledListBox`*** component allows the user to select an item from a drop-down list. We continue our discussion with a look at the ***`ScrolledText`*** component that allows a user to manipulate multiple lines of text. We also discuss menus; **`Pmw`** class **`MenuBar`** creates a component that helps a user organize a menu.

We also introduce more **`Tkinter`** classes. We use **`Tkinter`** class ***`Menu`*** to create popup menus—context-sensitive menus that typically appear when the user right clicks on components that have popup menus. Finally, we discuss the **`Tkinter`** ***`Canvas`*** component for displaying and manipulating text, images, lines and shapes. There are many GUI components and toolkits available to Python programmers, so we end this chapter with a description of several other toolkits.

11.2 Overview of `Pmw`

Python Megawidgets (**`Pmw`**) is a collection of useful GUI components built using module **`Tkinter`**. Each **`Pmw`** component combines one or more **`Tkinter`** components to create a useful, more complex component. Each **`Tkinter`** component can be referred to as a *subcomponent* of the **`Pmw`** component. For example, the **`Pmw`** ***`ComboBox`*** component combines an **`Entry`** component and a **`Listbox`** component to create a more complex component that enables users to select an item from a **`Listbox`** and edit the selection in an **`Entry`**.

Each subcomponent of a **`Pmw`** component can be configured independently—the appearance and functionality of the subcomponent can be modified. **`Pmw`** options have names of the form *subcomponent_option*, and the programmer configures a **`Pmw`** component by setting values for these options. Each component can be configured by passing option values in the constructor call either when the component is created or at a later time by passing option values in a call to method ***`configure`***. For example, the following

statement creates a **ScrolledListBox** **Pmw** component and configures the **Listbox** subcomponent with a height of 3:

```
Pmw.ScrolledListBox( self, listbox_height = 3 )
```

The following line configures the height of the **text** component in an existing **Pmw** ***TextDialog*** component

```
textdialog.configure( text_height = 10 )
```

Although **Pmw** extends the functionality of the **Tkinter** module by providing additional components, **Pmw** is not packaged with Python. To download the product, visit **pmw.sourceforge.net**. For installation instructions, visit the Deitel & Associates Web site at **www.deitel.com**.

11.3 ScrolledListbox Component

A *list box* (sometimes called a *drop-down list*) provides a list of items from which the user can select. **Tkinter** class ***Listbox*** (a derived class of **Widget**) implements list boxes.

In some cases, the number of items in a list prevents the list from being displayed entirely on the screen. In such cases, it is desirable to allow the user to scroll through the list. Scrolling can be implemented by configuring a **Scrollbar** and a **Listbox** to work together. However, **Pmw** offers a simpler option, the ***ScrolledListBox*** *megawidget*.

Figure 11.1 uses the **ScrolledListBox** component to provide a list of four image filenames. When the user selects an image filename, the program displays the corresponding image in a **Label**. The screen captures show the **ScrolledListBox** list after a selection.

```
# Fig. 11.1: fig11_01.py
# ScrolledListBox used to select image.

from Tkinter import *
import Pmw

class ImageSelection( Frame ):
   """List of available images and an area to display them"""

   def __init__( self, images ):
      """Create list of PhotoImages and Label to display them"""

      Frame.__init__( self )
      Pmw.initialise()
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Select an image" )

      self.photos = []

      # add PhotoImage objects to list photos
      for item in images:
         self.photos.append( PhotoImage( file = item ) )
```

Fig. 11.1 **ScrolledListBox** used to select an image. (Part 1 of 2.)

```
23
24         # create scrolled list box with vertical scrollbar
25         self.listBox = Pmw.ScrolledListBox( self, items = images,
26            listbox_height = 3, vscrollmode = "static",
27            selectioncommand = self.switchImage )
28         self.listBox.pack( side = LEFT, expand = YES, fill = BOTH,
29            padx = 5, pady = 5 )
30
31         self.display = Label( self, image = self.photos[ 0 ] )
32         self.display.pack( padx = 5, pady = 5 )
33
34      def switchImage( self ):
35         """Change image in Label to current selection"""
36
37         # get tuple containing index of selected list item
38         chosenPicture = self.listBox.curselection()
39
40         # configure label to display selected image
41         if chosenPicture:
42            choice = int( chosenPicture[ 0 ] )
43            self.display.config( image = self.photos[ choice ] )
44
45   def main():
46      images = [ "bug1.gif", "bug2.gif",
47                 "travelbug.gif", "buganim.gif" ]
48      ImageSelection( images ).mainloop()
49
50   if __name__ == "__main__":
51      main()
```

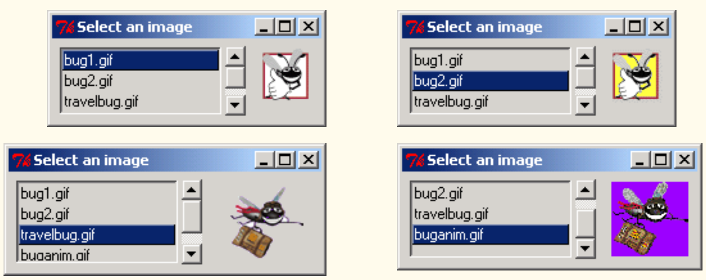


Fig. 11.1 **`ScrolledListBox`** used to select an image. (Part 2 of 2.)

Line 5 imports module **`Pmw`**. In line 14, function ***`Pmw.initialise`*** initializes **`Pmw`**. The call to function **`initialise`** enables the program to access the full functionality of the **`Pmw`** module.

Testing and Debugging Tip 11.1

*A program that uses module **`Pmw`** but does not invoke **`Pmw.initialise`** is not able to access the full functionality of module **`Pmw`**.*

Method **`main`** (lines 45–48) creates a list of image filenames, **`images`**, that the program passes to the constructor method of class **`ImageSelection`** (lines 7–43). Lines 21–22 create a list of **`PhotoImage`** instances from the filenames in **`images`**. Lines 25–

27 create a new **ScrolledListBox** component called **listBox**. The ***items*** option contains the list of items to be displayed in **listBox**. When the user selectes an entry in **listBox** with the left-mouse button, the method assigned to ***selectioncommand*** (**switchImage**) executes.

Note that the ***vscrollmode*** option for **listBox** is set to **"static"** (line 26). This setting ensures that the **vertscrollbar** subcomponent of the **ScrolledListBox** (a **Tkinter Scrollbar**) is always present. Other possible values are **"dynamic"** (which displays the **vertscrollbar** only if necessary) and **"none"** (which never displays the **vertscrollbar**). The default value is **"dynamic"**.

Line 31 creates a **Label** to display the selected image. By default, the label contains the image of the first item in the list. When the user selects an item in **listBox**, method **switchImage** (lines 34–43) changes the image. The call to **ScrolledListBox** method **curselection** (line 38) returns a tuple that contains one string. This string corresponds to the index of the user-selected listbox item. For example, if the user selects the **bug1.gif** image (the first image in the list), method **curselection** returns (**"0"**). If the tuple is not empty, **Tkinter** method **config** changes the **Label** component's **image** attribute to the user-selected image. The **ScrolledListBox** component also provides method ***getcurselection*** that returns a tuple of the currently selected values, rather than indices of those values.

By default, the user can select only one option in a **ScrolledListbox** component. A *multiple-selection list* enables the user to select several items from a **ScrolledListbox**. A **ScrolledListbox**'s ***listbox_selectmode*** option determines how many items a user may select. Possible values are **SINGLE**, **BROWSE** (default), **MULTIPLE** and **EXTENDED**. Value **SINGLE** allows the user to select one item at a time. Value **BROWSE** is the same as **SINGLE**, except that the user also may move the selection by dragging the mouse, rather than simply clicking an item. Value **MULTIPLE** allows the user to select multiple options, by clicking on multiple values. Value **EXTENDED** is similar to **BROWSE**, except that dragging the mouse selects multiple values. To select a contiguous range of items in a multiple-selection list, select the first item then press the *Shift* key while selecting the last item in the range. To select multiple, nonconsecutive items, press the *Ctrl* key while selecting each item. To deselect an item, hold the *Ctrl* key while clicking the item a second time.

A multiple-selection list does not have a specific event associated with making multiple selections. Normally, an *external event* generated by another GUI component (e.g., a **Button**) specifies when the multiple selections in a **ScrolleListbox** should be processed. We illustrate an example of an external event in the next section.

Look-and-Feel Observation 11.1

Often an external event determines when a program should process the selected items in a multiple-selection ***ScrolledListBox****.*

11.4 ScrolledText Component

Tkinter ***Text*** components provide areas for manipulating multiple lines of text. **Pmw** defines a ***ScrolledText*** component, which is a scrolled **Tkinter Text** component. Figure 11.2 contains two **ScrolledText** components—one displays text that the user can select and the other displays the text the user selected in the first **ScrolledText** compo-

nent. Sometimes, no event types are bound for a **ScrolledText**. Instead, an external event, (i.e., an event generated by a different GUI component) indicates when to process the text in a **ScrolledText** component. For example, many graphical e-mail programs provide a **Send** button to send the text of the message to the recipient. In this program, a button generates the external event that determines when the program copies the selected text in the left **ScrolledText** component into in the right **ScrolledText** component.

```python
# Fig. 11.2: fig11_02.py
# Copying selected text from one text area to another.

from Tkinter import *
import Pmw

class CopyTextWindow( Frame ):
   """Demonstrate ScrolledTexts"""

   def __init__( self ):
      """Create two ScrolledTexts and a Button"""

      Frame.__init__( self )
      Pmw.initialise()
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "ScrolledText Demo" )

      # create scrolled text box with word wrap enabled
      self.text1 = Pmw.ScrolledText( self,
         text_width = 25, text_height = 12, text_wrap = WORD,
         hscrollmode = "static", vscrollmode = "static" )
      self.text1.pack( side = LEFT, expand = YES, fill = BOTH,
         padx = 5, pady = 5 )

      self.copyButton = Button( self, text = "Copy >>>",
         command = self.copyText )
      self.copyButton.pack( side = LEFT, padx = 5, pady = 5 )

      # create uneditable scrolled text box
      self.text2 = Pmw.ScrolledText( self, text_state = DISABLED,
         text_width = 25, text_height = 12, text_wrap = WORD,
         hscrollmode = "static", vscrollmode = "static" )
      self.text2.pack( side = LEFT, expand = YES, fill = BOTH,
         padx = 5, pady = 5 )

   def copyText( self ):
      """Set the text in the second ScrolledText"""

      self.text2.settext( self.text1.get( SEL_FIRST, SEL_LAST ) )

def main():
   CopyTextWindow().mainloop()

if __name__ == "__main__":
   main()
```

Fig. 11.2 Text copied from one component to another. (Part 1 of 2.)

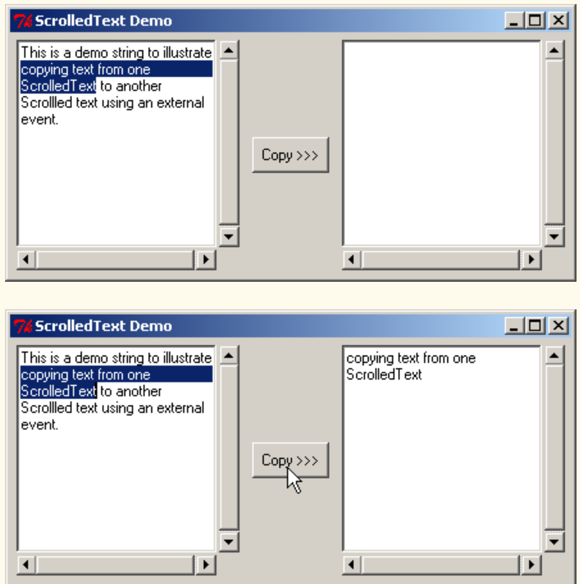


Fig. 11.2 Text copied from one component to another. (Part 2 of 2.)

Lines 19–23 create and pack the first **`ScrolledText`** component—**`text1`**—with a 25-column and 12-row **`Text`** subcomponent. The **`ScrolledText`** component's ***`text_wrap`*** option determines whether text lines that are too long to display in the component wrap. Value **`NONE`** (default) does not continue text on the next line and displays only the text that fits in the component's width. Value **`CHAR`** splits the text across multiple lines at the character location where the text becomes too long for the component. Value **`WORD`** is similar to value **`CHAR`**, except that the component splits the text on word boundaries (i.e., whitespace characters such as tabs and spaces). This last value enables word-wrapping, a common feature in many text editors.

Look-and-Feel Observation 11.2

To provide automatic word-wrap functionality for a ***`Text`*** *component, specify the* ***`text_wrap`*** *option as* ***`WORD`*** *rather than* ***`NONE`***.

Lines 25–27 create and pack **`copyButton`** and bind callback method **`copyText`**. Lines 30–34 create and pack the second **`ScrolledText`** component, **`text2`**. Line 30 sets **`text2`**'s **`text_state`** option to **`DISABLED`**, rendering the text area uneditable by disabling calls to **`insert`** and **`delete`** for the component.

When the user clicks **`copyButton`**, method **`copyText`** (lines 36–39) executes. This method retrieves the user-entered text from **`text1`** by invoking the component's method ***`get`***. Method **`get`** takes two arguments that specify the range of text to retrieve from the component. Line 39 retrieves the **`text1`**'s selected text by specifying a range that starts at the beginning of the selection (**`SEL_FIRST`**) and stops at the end of the selection (**`SEL_LAST`**). Method **`settext`** deletes the current text in the component and inserts the text the method receives as an argument. In this case, method **`settext`** inserts text returned by method **`get`** into **`text2`**. If the user has not selected any text, the program

raises a ***TclError*** exception and displays the error in an error dialog. We discussed exceptions briefly in Chapter 7, Object-Based Programming. In Chapter 12, Exception Handling, we discuss in detail how to handle exceptions (e.g., to prevent the program from displaying the error dialog).

11.5 MenuBar Component

Menus are an integral part of GUIs because they contain a list of actions, which, when selected by users, are performed by the applications. Menus simplify the appearances of graphical user interfaces by not "cluttering" the GUI with extra components (buttons, links, etc.). Simple **Tkinter** GUIs create menus with **Menu** components. Module **Pmw** includes class **MenuBar**, which contains the methods necessary to manage a *menu bar*, a container for menus.

Look-and-Feel Observation 11.3

Menus simplify GUIs by reducing the number of components the user views at one time.

A *menu item* is a GUI component inside a menu that performs an action when selected by a user. Menu items can be of different forms. The ***command*** menu item initiates an action. When a user selects a **command** menu item, the application invokes the item's callback method. The ***checkbutton*** menu item can be toggled on or off. When a user selects a **checkbutton** menu item, a checkmark appears to the left of the menu item. A user can select multiple **checkbutton**s (i.e, they are not mutually exclusive). Selecting a checked **checkbutton** removes the checkmark.

The ***radiobutton*** menu item is another menu item that can be toggled on or off. When multiple **radiobutton** menu items are grouped together, a user can select only one item from each **radiobutton**-menu-item group. After selecting a **radiobutton** menu item, a checkmark appears to the left of the menu item. When a user selects another **radiobutton** menu item from the same group, the application removes the checkmark from the previously selected menu item. Like **radioButton**s (discussed in Chapter 10, Graphical User Interface Components: Part 1), **radiobutton** menu items are grouped logically by a shared variable.

The ***separator*** menu item is a horizontal line in a menu. The **cascade** menu item is a *submenu* (or *cascade menu*) that provides more menu items from which the user can select.

Look-and-Feel Observation 11.4

The **separator** *menu item can be used to group related menu items.*

A menu bar contains menu items and submenus. When a menu is clicked, the menu expands to show its list of menu items and submenus. Clicking a menu item generates an event. Figure 11.3 provides menus and menu items that enable a user to change the properties of a line of text. The program also introduces *balloons* (also called a *tool-tips*) that display decriptions of menus and menu items. When the user moves the mouse cursor over a menu or menu item with a balloon, the program displays a specified help message.

Line 20 creates **myBalloon**—a **Pmw Balloon** component. Lines 21–23 create and pack a **MenuBar** component **choices**. Option **balloon** specifies a **Balloon** compo-

nent that is attached to the menubar. Lines 26–34 build the program's menu bar. Method ***addmenu*** (line 26) adds a new menu to **choices**. The method's first argument (**"File"**) is the menu name. The second argument (**"Exit"**) contains the text that appears in the menu's balloon. When the user places the mouse cursor over the **File** menu, the program displays this text in a floating label next to the cursor.

```
1  # Fig. 11.3: fig11_03.py
2  # MenuBars with Balloons demonstration.
3
4  from Tkinter import *
5  import Pmw
6  import sys
7
8  class MenuBarDemo( Frame ):
9     """Create window with a MenuBar"""
10
11    def __init__( self ):
12       """Create a MenuBar with items and a Canvas with text"""
13
14       Frame.__init__( self )
15       Pmw.initialise()
16       self.pack( expand = YES, fill = BOTH )
17       self.master.title( "MenuBar Demo" )
18       self.master.geometry( "500x200" )
19
20       self.myBalloon = Pmw.Balloon( self )
21       self.choices = Pmw.MenuBar( self,
22          balloon = self.myBalloon )
23       self.choices.pack( fill = X )
24
25       # create File menu and items
26       self.choices.addmenu( "File", "Exit" )
27       self.choices.addmenuitem( "File", "command",
28          command = self.closeDemo, label = "Exit" )
29
30       # create Format menu and items
31       self.choices.addmenu( "Format", "Change font/color" )
32       self.choices.addcascademenu( "Format", "Color" )
33       self.choices.addmenuitem( "Format", "separator" )
34       self.choices.addcascademenu( "Format", "Font" )
35
36       # add items to Format/Color menu
37       colors = [ "Black", "Blue", "Red", "Green" ]
38       self.selectedColor = StringVar()
39       self.selectedColor.set( colors[ 0 ] )
40
41       for item in colors:
42          self.choices.addmenuitem( "Color", "radiobutton",
43             label = item, command = self.changeColor,
44             variable = self.selectedColor )
45
```

Fig. 11.3 **MenuBar**s created with **Balloon**s. (Part 1 of 3.)

```python
      # add items to Format/Font menu
      fonts = [ "Times", "Courier", "Helvetica" ]
      self.selectedFont = StringVar()
      self.selectedFont.set( fonts [ 0 ] )

      for item in fonts:
         self.choices.addmenuitem( "Font", "radiobutton",
            label = item, command = self.changeFont,
            variable = self.selectedFont )

      # add a horizontal separator in Font menu
      self.choices.addmenuitem( "Font", "separator" )

      # associate checkbutton menu item with BooleanVar object
      self.boldOn = BooleanVar()
      self.choices.addmenuitem( "Font", "checkbutton",
         label = "Bold", command = self.changeFont,
         variable = self.boldOn )

      # associate checkbutton menu item with BooleanVar object
      self.italicOn = BooleanVar()
      self.choices.addmenuitem( "Font", "checkbutton",
         label = "Italic", command = self.changeFont,
         variable = self.italicOn )

      # create Canvas with text
      self.display = Canvas( self, bg = "white" )
      self.display.pack( expand = YES, fill = BOTH )

      self.sampleText = self.display.create_text( 250, 100,
         text = "Sample Text", font = "Times 48" )

   def changeColor( self ):
      """Change the color of the text on the Canvas"""

      self.display.itemconfig( self.sampleText,
         fill = self.selectedColor.get() )

   def changeFont( self ):
      """Change the font of the text on the Canvas"""

      # get selected font and attach size
      newFont = self.selectedFont.get() + " 48"

      # determine which checkbutton menu items selected
      if self.boldOn.get():
         newFont += " bold"

      if self.italicOn.get():
         newFont += " italic"

      # configure sample text to be displayed in selected style
      self.display.itemconfig( self.sampleText, font = newFont )
```

Fig. 11.3 `MenuBar`s created with `Balloon`s. (Part 2 of 3.)

```
99
100    def closeDemo( self ):
101       """Exit the program"""
102
103       sys.exit()
104
105 def main():
106    MenuBarDemo().mainloop()
107
108 if __name__ == "__main__":
109    main()
```

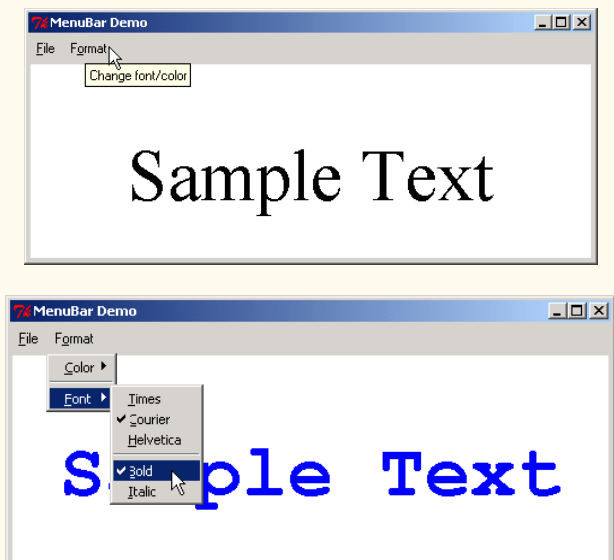


Fig. 11.3 `MenuBar`s created with `Balloon`s. (Part 3 of 3.)

Lines 27–28 invoke method ***`addmenuitem`*** to insert a **`command`** menu item in the **File** menu. This method requires two arguments—the name of the menu to which the item belongs and the menu item's type. This example adds the **Exit** menu item to the **File** menu. The method's keyword argument **`label`** specifies the menu item's text. The keyword argument **`command`** specifies the menu item's callback. When the user selects menu item **Exit** from the **File** menu, callback method **`closeDemo`** (lines 100–103) exits the program.

Line 31 adds menu **Format** to the **`choices`** menubar. Method ***`addcascademenu`*** (line 34) adds a submenu to an existing menu. The method requires two arguments—the name of the menu to which the submenu belongs and the submenu's text. When the user opens the **Format** menu and selects **Color**, the program displays the **Color** submenu. Lines 33–34 add a **`separator`** menu item and a **Font** submenu to menu **Format**. The **`separator`** menu item is a line dividing the **Color** and **Font** submenus.

Look-and-Feel Observation 11.5

Menu items appear in the order in which they are added. Be sure to add them in the correct order.

Look-and-Feel Observation 11.6

Menus normally appear from left to right in the order that they are added.

Line 37 defines the list of color choices for the sample text. Lines 38–39 create **StringVar selectedColor** and initialize it to the first element of the list of color choices. Lines 41–44 add a **radiobutton** menu item to the **Color** submenu for each item in a list of colors. Note that each **radiobutton** menu item shares the same callback method (**changeColor**) and the same variable (**selectedColor**). When the user selects an item, **selectedColor**'s value changes to the item's text value and method **changeColor** is invoked. Variable **selectedColor** is shared by the radiobutton menu items in the group.

Lines 51–54 add a **radiobutton** menu item for each item in a list of fonts to the **Format** menu's **Font** submenu. Each **radiobutton** menu item shares the same callback method (**changeFont**) and the same variable (**selectedFont**).

Line 57 adds a **separator** menu item to the **"Font"** submenu. Lines 60–69 then add **"Bold"** and **"Italic" checkbutton** menu items to the **Font** submenu. Lines 60 and 66 create two **BooleanVar** variables to represent whether these menue items are checked or unchecked). These values are passed to method **addmenuitem** through its **variable** keyword parameter. Although both **checkbutton** menu items share the same callback method (**changeFont**), they each have a different **BooleanVar** variable. The menu items' **BooleanVar** variables serve the same purpose as in **Tkinter Checkbutton** components. When the user selects the menu item, the **BooleanVar**'s value changes to 1. When the user deselects the menu item, the **BooleanVar**'s value changes to 0.

Lines 72–73 create and pack **display**—a **Tkinter *Canvas*** with a white background on which a program can display text, lines and shapes. A **Canvas** displays a *canvas item*—an object, like a string or a shape, that is drawn on the **Canvas** component. Each **Canvas** has a method that corresponds to a canvas item. Each of these methods creates a canvas item and adds it to the **Canvas**. For example, method **create_text** (lines 75–76) creates a canvas text item. This method draws the text **"Sample Text"** onto **display** in the font (**"Times 48"**) specified by keyword parameter **font**. We discuss **Canvas** components in more detail in Section 11.7.

When the user selects a **Color** menu item, method **changeColor** (lines 78–82) configures **sampleText** to be filled (colored) with the value of **selectedColor**. Method ***itemconfig*** configures items on **Canvas**. Lines 77–78 set the color of **sampleText** to the selected color by specifying option ***fill***.

When the user selects a **radiobutton** menu item in the **Font** submenu of the **Format** menu, method **changeFont** (lines 84–98) changes the font of **sampleText**. Line 98 retrieves the desired font name from **selectedFont**. Lines 91–95 determine whether any **checkbutton** menu items of the **Font** submenu are selected. If so, the program appends the specified style to the font name. Line 92 then updates the text with the specified font.

11.6 Popup Menus

Many of today's computer applications provide *context-sensitive popup menus*. Such menus can be created easily with **Tkinter** class **Menu**. These menus provide options that

are specific to the component for which the *popup trigger event* was generated. On most systems, the popup trigger event occurs when the user presses and releases the right mouse button. However, with **Tkinter**, a popup trigger event must be specified by binding a callback to the desired trigger for a component.

Figure 11.4 creates a **Menu** that allows the user to select one of three colors as the background color of the **Frame**. When the user clicks the right mouse button on the **Frame**, the program displays a popup menu containing a list of colors. If the user selects one of the **radiobutton** menu items that represents a color, the program changes the background color of the **Frame**.

```
# Fig. 11.4: fig11_04.py
# Popup menu demonstration.

from Tkinter import *

class PopupMenuDemo( Frame ):
   """Demonstrate popup menus"""

   def __init__( self ):
      """Create a Menu but do not add it to the Frame"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Popup Menu Demo" )
      self.master.geometry( "300x200" )

      # create and pack frame with initial white background
      self.frame1 = Frame( self, bg = "white" )
      self.frame1.pack( expand = YES, fill = BOTH )

      # create menu without packing it
      self.menu = Menu( self.frame1, tearoff = 0 )

      colors = [ "White", "Blue", "Yellow", "Red" ]
      self.selectedColor = StringVar()
      self.selectedColor.set( colors[ 0 ] )

      for item in colors:
         self.menu.add_radiobutton( label = item,
            variable = self.selectedColor,
            command = self.changeBackgroundColor )

      # popup menu on right-mouse click
      self.frame1.bind( "<Button-3>", self.popUpMenu )

   def popUpMenu( self, event ):
      """Add the Menu to the Frame"""

      self.menu.post( event.x_root, event.y_root )

```

Fig. 11.4 Popup menu implementation. (Part 1 of 2.)

```
41     def changeBackgroundColor( self ):
42        """Change the Frame background color"""
43
44        self.frame1.config( bg = self.selectedColor.get() )
45
46  def main():
47     PopupMenuDemo().mainloop()
48
49  if __name__ == "__main__":
50     main()
```

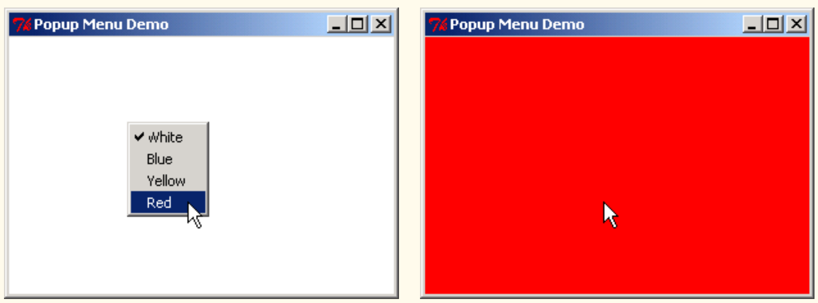


Fig. 11.4 Popup menu implementation. (Part 2 of 2.)

The **Frame** constructor's **bg** option is a string that specifies the **Frame**'s background color. Lines 18–19 create and pack **frame1** with a white background. Line 22 creates a **Tkinter Menu** component called **menu**. Note that **Menu**'s **tearoff** option is set to 0. This setting removes the dashed separator line that is, by default, the first entry in a **Menu**. Lines 28–31 add a **radiobutton** menu item to **menu** for each item in a list of colors. Each **radiobutton** menu item has the same callback method (**changeBackground-Color**) and the same variable (**selectedColor**).

Line 34 binds method **popUpMenu** to a right-mouse click (**<Button-3>**) for **frame1**. When the user right-clicks in **frame1**, the **popUpMenu** callback (lines 36–39) executes. Line 39 calls **Menu** method **post**, which displays a **Menu** at a given position. This method accepts two arguments that correspond to the position on the top-level component at which the menu is displayed. Event attributes **x_root** and **y_root** contain the coordinates of the mouse cursor when the event was triggered.

When a user selects one of the **radiobutton** menu items, method **changeBack-groundColor** executes. This method (lines 41–44) calls the **config** method of **frame1**, specifying the new **bg** to be the value of **selectedColor** (line 44). This method call changes **frame1**'s background color.

11.7 Canvas Component

Figure 11.3 used a ***Canvas*** to display formatted text. **Canvas** is a **Tkinter** component that displays text, images, lines and shapes. **Canvas** inherits from **Widget**. By default, a **Canvas** is blank. To display items on a **Canvas**, a program creates *canvas items*. New items are drawn on top of existing items unless otherwise specified.

Figure 11.5 uses the **<B1-Motion>** event and a **Canvas** to create a simple drawing program. The user draws pictures by dragging the mouse cursor over a **Canvas**.

```
# Fig. 11.5: fig11_05.py
# Canvas paint program.

from Tkinter import *

class PaintBox( Frame ):
   """Demonstrate drawing on a Canvas"""

   def __init__( self ):
      """Create Canvas and bind paint method to mouse dragging"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "A simple paint program" )
      self.master.geometry( "300x150" )

      self.message = Label( self,
         text = "Drag the mouse to draw" )
      self.message.pack( side = BOTTOM )

      # create Canvas component
      self.myCanvas = Canvas( self )
      self.myCanvas.pack( expand = YES, fill = BOTH )

      # bind mouse dragging event to Canvas
      self.myCanvas.bind( "<B1-Motion>", self.paint )

   def paint( self, event ):
      """Create an oval of radius 4 around the mouse position"""

      x1, y1 = ( event.x - 4 ), ( event.y - 4 )
      x2, y2 = ( event.x + 4 ), ( event.y + 4 )
      self.myCanvas.create_oval( x1, y1, x2, y2, fill = "black" )

def main():
   PaintBox().mainloop()

if __name__ == "__main__":
   main()
```



Fig. 11.5 **Canvas** paint program.

Lines 17–19 create and pack a **Label** with user instructions. Lines 22–23 create and pack **Canvas** instance **myCanvas**. Line 26 binds the mouse-drag event (**<B1-Motion>**) for the canvas to method **paint** (lines 28–33). When the user moves the mouse while holding down the left button, method **paint** executes. This method draws an oval on the **Canvas myCanvas**. **Canvas** method ***create_oval*** creates an ***oval Canvas** item* with a radius of 4 and a fill color of **"black"** centered at the current mouse cursor position (line 33).

11.8 Scale Component

The ***Scale*** component enables the user to select from a range of integer values. Class **Scale** inherits from **Widget**. Figure 11.6 shows a horizontal **Scale** with *numeric values* and a *slider* that allows the user to select a value.

Scales have either a *horizontal orientation* or a *vertical orientation*. On a horizontal **Scale**, the minimum value is at the extreme left and the maximum value is at the extreme right of the **Scale**. On a vertical **Scale**, the minimum value is at the extreme top and the maximum value is at the extreme bottom of the **Scale**.

Figure 11.7 enables the user to specify the size of a circle drawn on a **Canvas** by using a **Scale** component. The diameter of the circle is controlled with a horizontal **Scale**. The radius changes when the user interacts with the **Scale**.

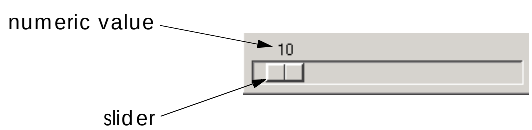


Fig. 11.6 Horizontal **Scale**.

```
# Fig. 11.7: fig11_07.py
# Scale used to control the size of a circle.

from Tkinter import *

class ScaleDemo( Frame ):
   """Demonstrate Canvas and Scale"""

   def __init__( self ):
      """Create Canvas with a circle controlled by a Scale"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Scale Demo" )
      self.master.geometry( "220x270" )

      # create Scale
      self.control = Scale( self, from_ = 0, to = 200,
         orient = HORIZONTAL, command = self.updateCircle )
```

Fig. 11.7 **Scale** used to control the size of a circle on a **Canvas**. (Part 1 of 2.)

```
20          self.control.pack( side = BOTTOM, fill = X )
21          self.control.set( 10 )
22
23          # create Canvas and draw circle
24          self.display = Canvas( self, bg = "white" )
25          self.display.pack( expand = YES, fill = BOTH )
26
27       def updateCircle( self, scaleValue ):
28          """Delete the circle, determine new size, draw again"""
29
30          end = int( scaleValue ) + 10
31          self.display.delete( "circle" )
32          self.display.create_oval( 10, 10, end, end,
33             fill = "red", tags = "circle" )
34
35    def main():
36       ScaleDemo().mainloop()
37
38    if __name__ == "__main__":
39       main()
```

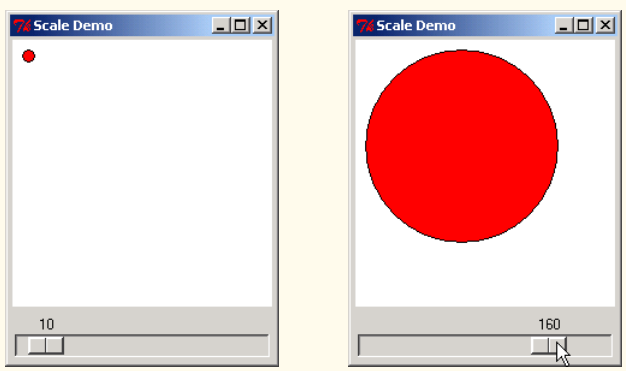


Fig. 11.7 **Scale** used to control the size of a circle on a **Canvas**. (Part 2 of 2.)

Lines 18–20 create and pack **control**, the **Scale** used to change the size of the circle. The constructor's ***orient*** option (***HORIZONTAL*** or ***VERTICAL***) determines whether the new **Scale** instance has a horizontal or vertical orientation. Options ***from_*** and ***to*** specify the **Scale** component's minimum and maximum values, respectively. The option values in lines 18–19 create a horizontal **Scale** with a minimum value of 0 and a maximum value of 200. The **Scale**'s callback is method **updateCircle**, which executes when the user moves the slider to change the numerical value. Note that although nothing is drawn on **display** in **__init__**, the circle appears on **display** when the program starts. This is because when the **Scale** is created, its callback method (**updateCircle**) is invoked. Line 21 sets **control**'s value to 10, so that when the program starts, a circle of diameter 10 appears on the screen. Lines 24–25 create and pack **display**, a **Canvas** with a white background.

When the user drags the slider, method ***updateCircle*** (lines 27–33) executes. The callback accepts as an argument the current value of the scale, represented as a string. Line 30 converts this value to an integer, adds 10 to it and stores the value in variable **end**.

Canvas method ***delete*** (line 31) deletes the old circle before drawing a new one. Method **delete** accepts one argument—either an *item handle* or a *tag*. Item handles are integer values that identify a newly drawn item. A tag is a name that can be attached to a canvas item at creation. To attach a tag to a canvas item, pass a string value to the ***tags*** option of the item's **create** method.

Method ***create_oval*** (lines 32–33) draws an oval with coordinates (**10**, **10**, **end**, **end**), specifying option **fill** to be **"red"** and option **tags** to be **"circle"**. The coordinates specify points on the oval's bounding rectangle. **Canvas** method **create_*item*** allows the user to create the following items by substituting their names for ***item***—arc, line, oval, rectangle, polygon, image, bitmap, text and window.

11.9 Other GUI Toolkits

Many different GUI Toolkits for Python exist. **PyGTK** (**www.daa.com.au/~james/pygtk**) provides an object-oriented interface for the *Gimp ToolKit (GTK)* component set (**www.gtk.org**). GTK is an advanced component set used primarily under the X Windows system (a graphics system providing a common interface for displaying windowed graphics). PyGTK is a part of GTK**+**, a Python toolkit for creating graphical user interfaces.

Another popular GUI toolkit is **wxPython** (**www.wxpython.org**)—a Python extension module that enables access to **wxWindows**, a GUI library written in C++. This toolkit currently supports Microsoft Windows and most of the Unix-like systems. Python module **wxPython** wraps around wxWindows, providing the interface to manipulate wxWindows classes and methods.

PyOpenGL (**pyopengl.sourceforge.net**) provides a Python interface to the OpenGL (**www.opengl.org**) library. OpenGL is one of the most widely used libraries designed for developing interactive two-dimensional and three-dimensional graphical applications. It is available under Microsoft Windows, MacOS and most Unix-like systems. **PyOpenGL** can be used with **Tkinter**, **wxPython** and other windowing libraries. Chapter 24, Multimedia, discusses module **PyOpenGL**.

SUMMARY

- The **Pmw** (Python Mega Widgets) toolkit provides high-level GUI components composed of **Tkinter** components.
- Megawidgets can also be configured for a particular use. The appearance and functionality of the components and their subcomponents can be modified.
- The components can be configured either during or after creation with method **configure**.
- In general, subcomponent options are named *subcomponent_option*.
- A list box provides a list of items from which the user can make a selection. List boxes are implemented with **Tkinter** class **Listbox**, which inherits from class **Widget**.
- Often, it is desirable to allow the user to scroll up and down a list. Scrolling can be achieved by creating a **Tkinter Scrollbar** and a **Listbox** separately and configuring them properly. Conveniently, **Pmw** provides a megawidget called **ScrolledListBox** that serves this purpose.
- Function **Pmw.initialise** initializes **Pmw**. This function call allows a list of top-level components to be maintained. This call also ensures that **Pmw** is notified after the destruction of a component.
- The **items** option contains the list of items that will be displayed in a **ScrolledListBox**.

- The method specified as a value for option **selectioncommand** executes each time an entry in a **ScrolledListBox** is selected.
- Setting the **vscrollmode** option for a **ScrolledListBox** to **"static"** ensures that the **vertscrollbar** subcomponent of the **ScrolledListBox** (a **Tkinter Scrollbar**) will always be present. Other possible values are **"dynamic"** (display the **vertscrollbar** only if necessary) and **"none"** (the **vertscrollbar** will never be displayed). The default value is **"dynamic"**.
- Method **curselection** returns a tuple of the indices of the currently selected items in a **ScrolledListBox**.
- The **ScrolledListBox** component also supports a **getcurselection** method that returns a tuple of the currently selected values, rather than the values' indices.
- By default, the user can select only one option in a **ScrolledListbox** component.
- A multiple-selection list enables the user to select many items from a **ScrolledListbox**.
- A **ScrolledListbox**'s **listbox_selectmode** option controls how many items a user may select. Possible values are **SINGLE**, **BROWSE** (default), **MULTIPLE** and **EXTENDED**. Value **SINGLE** allows the user to select only one item in the **ScrolledListbox** at a time. Value **BROWSE** is the same as **SINGLE**, except that the user also may move the selection by dragging the mouse, rather than simply clicking an item. Value **MULTIPLE** allows the user to select multiple options, by clicking on multiple values. Value **EXTENDED** acts like **BROWSE**, except that when the user drags the mouse, the user selects multiple values.
- A multiple-selection list does not have a specific event associated with making multiple selections. Normally, an external event generated by another GUI component specifies when the multiple selections in a **ScrolledListbox** should be processed.
- **Tkinter Text** components provide an area for manipulating multiple lines of text. **Pmw** defines a **ScrolledText** component, which is a scrolled **Tkinter Text**.
- Sometimes, no event types are bound for a **ScrolledText**. Instead, an external event indicates when the text in a **ScrolledText** should be processed.
- The **ScrolledText** component's **wrap** option controls the appearance of text lines that are too long to display in the component. Value **NONE** (default) for **wrap** means that the component truncates the line and displays only the text that fits in the component. Value **CHAR** for **wrap** means that the text is broken up when it becomes too long; the remainder of the text is displayed on the next line. Value **WORD** for **wrap** is similar to value **CHAR**, except that the component breaks the text on word boundaries. This last value enables word-wrapping, a common feature in many popular text editors.
- Setting a text subcomponent's state as **DISABLED** renders the text area uneditable by disabling calls to **insert** and **delete** for the component.
- The **ScrolledText** component's method **get** retrieves the user-entered text. Method **get** takes two arguments that specify the range of text to retrieve from the component. Constant **SEL_FIRST** specifies the beginning of the selection. Constant **SEL_LAST** specifies the end of the selection.
- Method **settext** deletes the current text in the component and inserts the specified text.
- Menus are an integral part of GUIs. Menus allow the user to perform actions without unnecessarily "cluttering" a graphical user interface with extra GUI components.
- Simple **Tkinter** GUIs create menus with **Menu** components. However, **Pmw** supplies class **MenuBar**, which contains the methods necessary to manage a menu bar, a container for menus.
- A menu item is a GUI component inside a menu that causes an action to be performed when selected. Menu items can be of different forms.

- A **command** menu item initiates an action. When the user selects a **command** menu item, the item's callback method is invoked.
- A **checkbutton** menu item can be toggled on or off. When a **checkbutton** menu item is selected, a check appears to the left of the menu item. When the **checkbutton** menu item is selected again, the check to the left of the menu item is removed.
- When multiple **radiobutton** menu items are assigned to the same variable, only one item in the group can be selected at a given time. When a **radiobutton** menu item is selected, a check appears to the left of the menu item. When another **radiobutton** menu item is selected, the check to the left of the previously selected menu item is removed.
- A **separator** menu item is a horizontal line in a menu that groups menu items logically.
- A **cascade** menu item is a submenu. A submenu (or cascade menu) provides more menu items from which the user can select.
- When a menu is clicked, the menu expands to show its list of menu items and submenus.
- Clicking a menu item generates an event.
- A balloon (also called a tool-tip) displays helpful text for menus and menu items. When the user moves the mouse cursor over a menu or menu item with a balloon, the program displays a specified help message.
- Option **balloon** specifies a **Balloon** component that is attached to the menu.
- Method **addmenu** of **Pmw** class **MenuBar** adds a new menu. The method's first argument contains the name of the menu; the second argument contains the text that appears in the menu's balloon. When the user places the mouse cursor over the menu, the program displays this text.
- Method **addmenuitem** of **Pmw** class **MenuBar** adds a menu item to a menu. This method requires two arguments: the name of the menu to which the item belongs and the menu item's type.
- **MenuBar** method **addmenuitem**'s keyword argument **label** specifies the menu item's text. Keyword argument **command** specifies the item's callback.
- Method **addcascademenu** of **Pmw** class **MenuBar** adds a submenu to an existing menu. The method requires two arguments: the name of the menu to which the submenu belongs and the submenu's text.
- Many of today's computer applications provide context-sensitive popup menus. These menus provide options that are specific to the component for which the popup trigger event was generated.
- Context menus be created easily with **Tkinter** class **Menu** (a subclass of **Widget**). A popup trigger event must be specified by binding a callback to the desired trigger for a component.
- The **Frame** constructor's **bg** option takes a string specifying the **Frame**'s background color.
- Setting a **Menu**'s **tearoff** option to 0 removes the dashed separator line that is, by default, the first entry in a **Menu**.
- **Menu** method **post** displays a **Menu** at a given position. This method accepts two arguments that correspond to the position on the top-level component at which the menu is displayed.
- The current mouse position is specified by the **x_root** and **y_root** attributes of the **Event** instance passed to an event handler.
- **Canvas** is a **Tkinter** component that displays text, images, lines and shapes. **Canvas** inherits from **Widget**.
- By default, a **Canvas** is blank. To display items on a **Canvas**, a program creates canvas items. New items are drawn on top of existing items unless otherwise specified.
- Adding canvas items to a **Canvas** displays something on the **Canvas**. Each canvas item has a corresponding **Canvas** method that creates the item and adds it to the canvas.

- Method **create_text** of class **Canvas** creates a canvas text item. **Canvas** method **create_oval** creates an **oval** **Canvas** item.
- Method **itemconfig** of class **Canvas** configures items on **Canvas**.
- Specifying a value for option **fill** sets the color of a canvas item.
- The **Scale** component enables the user to select from a range of integer values. Class **Scale** inherits from **Widget**. **Scale**s have either a horizontal orientation or a vertical orientation. For a horizontal **Scale**, the minimum value is at the extreme left and the maximum value is at the extreme right of the **Scale**. For a vertical **Scale**, the minimum value is at the extreme top and the maximum value is at the extreme bottom of the **Scale**.
- The **Scale** constructor's **orient** option (**HORIZONTAL** or **VERTICAL**) determines whether the new **Scale** instance has a horizontal or vertical orientation. Options **from_** and **to** specify the **Scale** component's minimum and maximum values. When the **Scale** is created, its callback method is invoked.
- Item handles are integer values that identify a newly drawn item.
- A tag is a name that can be attached to a canvas item when the item is created.
- **Canvas** method **delete** deletes a canvas item. Method **delete** accepts one argument—either an item handle or a tag.
- To attach a tag to a canvas item, pass a string value to the **tags** option of the item's **create** method.
- **Canvas** methods **create_item** allow the user to create the following items by substituting their names for **item**: arc, line, oval, rectangle, polygon, image, bitmap, text and window.
- **PyGTK** provides an object-oriented interface for the GTK component set (**www.gtk.org**). GTK is an advanced component set used primarily under the X Windows system (a graphics system providing a common interface for displaying windowed graphics).
- **wxPython** is a Python extension module that enables access of wxWindows. wxWindows is a GUI library written in C++. It currently supports Microsoft Windows and most of the Unix-like systems.
- **PyOpenGL** provides a Python interface to the OpenGL (**www.opengl.org**) library—one of the most widely used libraries designed for developing interactive two-dimensional and three-dimensional graphical applications. It is available under Microsoft Windows, MacOS and most Unix-like systems. **PyOpenGL** can be used with **Tkinter**, **wxPython** and other windowing libraries.

TERMINOLOGY

addcascademenu method of **MenuBar**
addmenu method of **MenuBar** component
addmenuitem method of **MenuBar**
balloon
balloon option of **MenuBar**
bg option of **Frame** component
BROWSE value of **listbox_selectmode** option of **ScrolledListbox**
Canvas component
cascade menu
cascade menu item
CHAR option of **wrap** option of **ScrolledText**
checkbutton menu item
command menu item
command option of **MenuBar**
configure method of **Pmw**
create_oval method of **Canvas**
curselection method of **ScrolledListBox**
"none" option of **vscrollmode** option of **ScrolledListBox**
"static" option of **vscrollmode** option of **ScrolledListBox**
"dynamic" value of **vscrollmode** option of **ScrolledListBox**
EXTENDED value of **listbox_selectmode** option of **ScrolledListbox**
external event
fill option of **Canvas**
font option of **Canvas**
from_ option of **Scale**

get method of **ScrolledText**
getcurselection method of **ScrolledListBox**
horizontal **Scale** component
HORIZONTAL value of **orient** option of **Scale**
item handle
itemconfig method of **Canvas**
items option of **ScrolledListBox**
Listbox component
menu
menu bar
Menu component
menu item
mnemonics
multiple-selection list
NONE value of **wrap** option of **ScrolledText**
orient option of **Scale**
Pmw.initialise function
popup trigger event
post method of **Menu** component
radiobutton menu item
Scale component
Scrollbar component
ScrolledListBox component
ScrolledText component
SEL_FIRST argument to method **get** of **ScrolledText**
SEL_LAST argument to method **get** of **ScrolledText**
listbox_selectmode option of **ScrolledListbox**
separator menu item
settext method of **ScrolledText**
SINGLE value of **listbox_selectmode** option of **ScrolledListbox**
tag
tags option of **create** method of **Canvas**
tearoff option of **Menu**
Text component
to option of **Scale**
tool-tip
traverseSpec option of **MenuBar**
variable keyword argument for method **addmenuitem**
vertical **Scale** component
VERTICAL value of **orient** option of **Scale**
vscrollmode option of **ScrolledListBox**
WORD option of **wrap** option of **ScrolledText**
word wrapping
wrap option of **ScrolledText** component

SELF-REVIEW EXERCISES

11.1 Fill in the blanks in each of the following:
a) **Tkinter** class __________, which inherits from class __________, implement list boxes.
b) If the **vscrollmode** of a vertical **ScrolledListBox** is set to __________, the scroll bar component will never be displayed.
c) A __________ enables the user to select many items from a list box.
d) Set **text_wrap** to __________ in a **ScrolledText** widget to enable word wrap.
e) When the user selects a __________ menu item, its callback function is invoked.
f) A __________ displays help text for menu items.
g) **Tkinter** component __________ displays text, images, lines and shapes.
h) The __________ component enables a user to select from a range of integer values.
i) __________ are integer values identifying an item drawn on a **Canvas**.
j) An __________ allows selection of a contiguous range of items in the list.

11.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) **Tkinter** cannot provide a scrollbar with a list.
b) By default, the scrollbar component of a **ScrolledListBox** is always displayed.
c) The **Pmw** component **ScrolledText** is a scrolled **Tkinter Text**.
d) **Tkinter Menu** components contain the methods necessary to manage a menu bar.
e) A **cascade** menu is a submenu that provides more items from which the user can select.
f) Method **addmenuitem** adds menus to a menu bar, which can contain menu items.
g) **Tkinter** class **Menu** can create context-sensitive popup menus.
h) The minimum and maximum value positions on a **Scale** can be specified by setting the **from_** and **to** options.

i) A **Scale** must be horizontal with the maximum value at the extreme right and the minimum value at the extreme left.
j) A **radiobutton** menu item can be toggled on and off.

ANSWERS TO SELF-REVIEW EXERCISES

11.1 a) **Listbox**, **Widget**. b) **"none"**. c) multiple-selection list. d) **WORD**. e) **command**. f) balloon. g) **Canvas**. h) **Scale**. i) Item handles. j) **EXTENDED Listbox**.

11.2 a) False. A **Tkinter Scrollbar** can be attached to a **Listbox** to create a scrollable list. b) False. By default, the scrollbar component of a **ScrolledListBox** is displayed only if it is necessary to navigate the list. c) True. d) False. Pmw class **MenuBar** contains the methods necessary to manage a menu bar. e) True. f) False. Method **addmenu** adds menus to a menu bar, which can contain menu items. g) True. h) True. i) False. A **Scale** may have either a horizontal or a vertical orientation. j) True.

EXERCISES

11.3 Modify Exercise 10.4. Allow the user to select a Fahrenheit temperature to be converted with a horizontal **Scale**. When the user interacts with the **Scale**, update the temperature conversion.

11.4 Rewrite the program of Fig. 11.2. Create a multiple-selection list of colors. Allow the user to select one or more colors and copy them to a **ScrolledText** component.

11.5 Write a program that allows the user to draw a rectangle by dragging the mouse on a **Canvas**. The drawing should begin when the user holds the left-mouse button down. With this button held down, the user should be able to resize the rectangle. The drawing ends when the user releases the left button. When the user next clicks on the **Canvas**, the rectangle should be deleted.

11.6 Modify Exercise 11.5. Allow the user to fill the rectangle with a color. Create a popup menu of possible colors. The popup menu should appear when the user presses the right-mouse button.

11.7 Write a menu designer program. The program allows the user to enter menu information and generates code to create that menu based on the user input. The GUI allows the user to enter the necessary input and displays the menu names in a **Pmw ScrolledListBox** as they are added. The program displays the generated code when the user has finished adding information. The program can be written with two distinct parts.
a) Class **Menus** creates a GUI that allows the user to enter a menu name or a menu name, menu item and callback function. The program should issue a warning with a dialog box if the user does not enter the specified information. The GUI provides two **Entry** components for the menu name and the menu item and a **Pmw ScrolledText** component for the callback function. The program generates code based on the user input that the user could execute to create the menu. The GUI should have an **Add** button whose callback adds the user input to the generated menu code, a **Clear** button whose callback resets the GUI and a **Finish** button that ends the program, displaying the generated menu code.
b) Class **MenusList** creates a single-selection list of the added menus. As the user adds menus, the **Pmw ScrolledListBox** should be updated with the new information. When the user selects a menu in the list, the menu items in that list are displayed in a **Pmw ScrolledText** component.

You may add extra error checking and special features. For instance, when the user selects a menu name in the **Pmw ScrolledListBox** display the menu name in the **Entry** component for menu names.

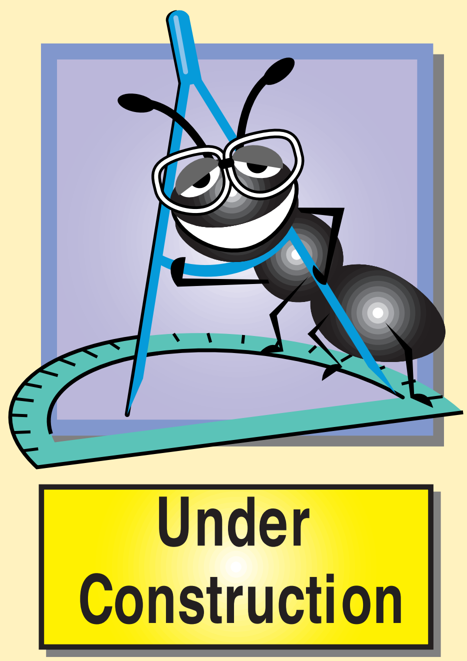
11.8 Modify Exercise 11.7 so that, as the user adds menus and menu items a sample menu displays and is updated with any new information.

12

Exception Handling

Objectives

- To understand exceptions and error handling.
- To use the **`try`** statement to delimit code in which exceptions may occur.
- To be able to **`raise`** exceptions.
- To use **`except`** clauses to specify exception handlers.
- To use the **`finally`** clause to release resources.
- To understand the Python exception class hierarchy.
- To understand Python's traceback mechanism.
- To create programmer-defined exceptions.



It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.
Franklin Delano Roosevelt

O! throw away the worser part of it,
And live the purer with the other half.
William Shakespeare

If they're running and they don't look where they're going
I have to come out from somewhere and catch them.
Jerome David Salinger

And oftentimes excusing of a fault
Doth make the fault the worse by the excuse.
William Shakespeare

Outline

12.1 Introduction

In this chapter, we introduce *exception handling*. An *exception* is an indication of a "special event" that occurs during a program's execution. The name "exception" indicates that, although the event can occur, the event occurs infrequently. Often, the special event is an error (e.g., dividing by zero or adding two incompatible types); sometimes, the special event is something else (e.g., the termination of a **`for`** loop). Exception handling enables programmers to create applications that can *handle* (or *resolve*) exceptions. In many cases, handling an exception allows a program to continue executing as if no problems were encountered. More severe problems may prevent a program from continuing normal execution. In such cases, the program can notify the user of the problem, then terminate in a controlled manner. The features presented in this chapter enable programmers to write programs that are clear, robust and more *fault tolerant*.

The style and details of exception handling in Python are based on the work of the creators of the Modula-3 programming language. The exception-handling mechanism is similar to that used in C# and Java.

We begin with an overview of exception-handling concepts, then demonstrate basic exception-handling techniques. The chapter then overviews the exception-handling class hierarchy.

Programs typically request and release resources (such as files on disk) during program execution. Often, these resources are in limited supply or can be used only by one program at a time. We demonstrate a part of the exception-handling mechanism that enables a program to use a resource, then guarantee that the program releases the resource for use by other programs.

The chapter continues with an explanation and example of **`traceback`** objects—the objects that Python creates when it encounters an exception. The chapter concludes with an example that shows programmers how to create and use their own exception classes.

12.2 Raising an Exception

Chapter 7, Classes and Data Abstraction, introduced the ***`raise`*** statement, to signal that a client had attempted to assign an invalid value to an object's attribute. The **`raise`** state-

ment indicates that an exception occurred (e.g., a function could not complete successfully). This is called *raising* (or sometimes *throwing*) *an exception*.

The simplest form of raising an exception consists of the keyword **`raise`**, followed by the name of the exception to raise. Exception names identify classes; Python exceptions are objects of those classes. When a **`raise`** statement executes, Python creates an object of the specified exception class. The **`raise`** statement also may specify arguments that initialize the exception object. To do so, follow the exception class name with a comma (**`,`**) and the argument (or a tuple of arguments). Programs can use an exception object's attributes to discover more information about the exception that occurred. The **`raise`** statement has many forms. Section 12.7 discusses another form of **`raise`** that specifies no exception name.

Testing and Debugging Tip 12.1

The arguments used to initialize an exception object can be referenced in an exception handler to perform an appropriate task.

Testing and Debugging Tip 12.2

An exception can be raised without passing arguments to initialize the exception object. In this case, knowledge that an exception of this type occurred normally provides sufficient information for the handler to perform its task.

Until now, we have seen only how a **`raise`** statement causes a program to terminate and print an error message. This chapter demonstrates how a program detects that an exception occurred (called *catching* an exception), then, based on that exception, takes appropriate action (called *handling* the exception). Catching and handling exceptions enables a program to know when an error has occurred, then to take actions to minimize the consequences of that error.

12.3 Exception-Handling Overview

The logic of a program frequently tests conditions that determine how program execution proceeds. Consider the following pseudocode:

Perform a task

If the preceding task did not execute correctly
    Perform error processing

Perform next task

If the preceding task did not execute correctly
    Perform error processing

…

This pseudocode begins by performing a task, then tests a condition to determine whether that task executed correctly. If not, error processing occurs. Otherwise, the pseudocode continues with the next task. Although this form of error handling may work, intermixing the logic of the program with the error-handling logic can make the program difficult to read, modify, maintain and debug—especially in large applications. In fact, if many of the potential problems occur infrequently, intermixing program logic and error handling can degrade the performance of the program, because the program must test extra conditions to determine whether the next task can be performed.

Exception handling enables programmers to remove error-handling code from the "main line" of the program's execution. This improves program clarity and enhances modifiability. Programmers can decide to handle whatever exceptions they choose—all types of exceptions, all exceptions of a certain type or all exceptions of a related type. Such flexibility reduces the likelihood that errors will be overlooked, thereby increasing a program's robustness.

Testing and Debugging Tip 12.3

Exception handling helps improve a program's fault tolerance. When it is easy to write error-processing code, programmers are more likely to use it.

With programming languages that do not support exception handling, programmers often delay writing error-processing code and sometimes simply forget to include it. This results in less robust software products. Python enables the programmer to deal with exception handling easily from the inception of a project. Still, the programmer must put considerable effort into incorporating an exception-handling strategy into software projects.

Software Engineering Observation 12.1

Incorporate your exception-handling strategy into a system from the inception of the design process. Adding effective exception handling after a system has been implemented is difficult.

Software Engineering Observation 12.2

In the past, programmers used many techniques to implement error-processing code. Exception handling provides a single, uniform technique for processing errors. This enables programmers working on large projects to understand each other's error-processing code.

The exception-handling mechanism also is useful for processing problems that occur when a program interacts with reusable software elements, such as functions, classes and modules. Rather than internally handling problems that occur, such software elements use exceptions to notify client code when problems occur. This enables programmers to implement error handling that is appropriate to each application.

Common Programming Error 12.1

Aborting a program component could leave a resource—such as a file or a network connection—in a state in which other programs are not able to acquire the resource. This is known as a "resource leak."

Performance Tip 12.1

When no exceptions occur, exception-handling code incurs little or no performance penalties. Thus, programs that implement exception handling operate more efficiently than programs that perform error handling throughout the program logic.

Software Engineering Observation 12.3

Complex applications normally consist of predefined software components (such as those defined in the Python standard library) and components specific to the application that use the predefined components. When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the predefined component cannot know in advance how each application will process a problem that occurs. Exception handling simplifies combining software components and having them work together effectively by enabling predefined components to communicate problems that occur to application-specific components, which can then process the problems in an application-specific manner.

Software Engineering Observation 12.4

Although it is possible to do so, exceptions often are not used explicitly for conventional flow of control. It is more difficult to keep track the of consequently larger number of exception cases, which makes programs difficult to read and maintain.

Exception handling is geared to situations in which the code that detects an error is unable to handle it. Such code *raises* or *throws an exception*. There is no guarantee that there will be an *exception handler*—code that executes when the program detects an exception—to process that kind of exception. If there is, the exception will be *caught* (detected) and *handled*. The result of an *uncaught exception* depends on whether the program is a GUI program or a *console* (non-GUI) program and on whether the program is running in interactive mode. In a non-GUI program, an uncaught exception simply causes the program to print an error message and terminate. When a GUI program detects an uncaught exception, the program displays the error message (either in the console or in a dialog box, depending on the GUI package) and the program continues execution. Although a GUI program continues execution after an uncaught exception, the program may fail to behave as expected, because of the error that caused the exception. When a program running in interactive mode detects an uncaught exception, the program displays an error message, terminates execution and displays the interactive Python prompt.

Python uses ***`try`*** statements to enable exception handling. The **`try`** statement encloses other statements that potentially cause exceptions. A **`try`** statement begins with keyword **`try`**, followed by a colon (**`:`**), followed by a suite of code in which exceptions may occur. The **`try`** statement may specify one or more ***`except`** clauses* that immediately follow the **`try`** suite. Each **`except`** clause specifies zero or more exception class names that represent the type(s) of exceptions that the **`except`** clause can handle. An **`except`** clause (also called an ***`except`** handler*) also may specify an identifier that the program can use to reference the exception object that was caught. The handler can use the identifier to obtain information about the exception from the exception object. An **`except`** clause that specifies no exception type is called an *empty **`except`** clause*. Such a clause catches all exception types. After the last **`except`** clause, an optional **`else`** clause contains code that executes if the code in the **`try`** suite raised no exceptions. If a **`try`** statement specifies no **`except`** clauses, the statement must contain a **`finally`** clause, which always executes, regardless of whether an exception occurs. We discuss each possible combination of clauses over the next several sections.

Common Programming Error 12.2

*It is a syntax error to write a **`try`** statement that contains **`except`** and **`finally`** clauses. The only acceptable forms are **`try`**/**`except`**, **`try`**/**`except`**/**`else`** and **`try`**/**`finally`**.*

When code in a program causes an exception, or when the Python interpreter detects a problem, the code or the interpreter *raises* (or *throws*) *an exception*. Some programmers refer to the point in the program at which an exception occurs as the *throw point*—an important location for debugging purposes (as we demonstrate in Section 12.7). Exceptions are objects of classes that inherit from class ***`Exception`***.[1] If an exception occurs in a **`try`**

1. Python exceptions also may be strings, to support programs that require earlier versions of the Python interpreter. For newer Python versions (greater than 1.5.2), the class-based exception-handling technique is preferred.

suite, the **`try`** suite *expires* (i.e., terminates immediately), and program control transfers to the first **`except`** handler (if there is one) following the **`try`** suite. Next, the interpreter searches for the first **`except`** handler that can process the type of exception that occurred. The interpreter locates the matching **`except`** by comparing the raised exception's type to each **`except`**'s exception type(s) until the interpreter finds a match. A match occurs if the types are identical or if the raised exception's type is a derived class of the handler's exception type. If no exceptions occur in a **`try`** suite, the interpreter ignores the exception handlers for the **`try`** statement and executes the **`try`** statement's **`else`** clause (if the statement specifies an **`else`** clause). If no exceptions occur, or if one of the **`except`** clauses successfully handles the exception, program execution resumes with the next statement after the **`try`** statement. If an exception occurs in a statement that is not in a **`try`** suite and that statement is in a function, the function containing that statement terminates immediately and the interpreter attempts to locate an enclosing **`try`** statement in a calling code—a process called *stack unwinding* (discussed in Section 12.7).

Python is said to use the *termination model of exception handling*, because the **`try`** suite that raises an exception expires immediately when that exception occurs.[2]

12.4 Example: `DivideByZeroError`

Let us consider a simple exception-handling example. The program in Fig. 12.1 uses **`try`**, **`except`** and **`else`** to detect and handle exceptions. The program prompts the user to enter two numbers that represent the numerator and denominator of a division. After the user enters the two numbers, the program calls function **`float`** on each user-entered string, to convert the user inputs to floating-point values. The program then attempts to divide the first value by the second value. If the user types 0 in response to the request for a denominator, an exception occurs when the program attempts to divide by zero. Also, if the user types a value that is not a number in response to either prompt, the program displays a message requesting that the user enter two numbers.

Before we discuss the program details, consider the sample outputs Fig. 12.1. The first shows a successful calculation in which the user inputs the numerator **`100`** and the denominator **`7`**. The output shows the result of the division. In the second output, the user enters the string **`"hello"`** at the second prompt. When the user presses *Enter* after typing the string, the program displays a message, indicating that the user must enter numbers. This occurs because **`float`** cannot convert a string argument to a floating-point value, so the function raises a ***`ValueError`*** exception. The program catches the exception and displays an appropriate message. The last output shows the result after an attempt to divide by zero. The Python interpreter itself tests for division by zero and raises a ***`ZeroDivisionError`*** exception if the denominator is zero. The program catches the exception and displays a message, indicating an attempt to divide by zero.

Let us consider the user interactions and flow of control that yield the results shown in the sample input/output dialogs. The user inputs values that represent the numerator and

2. Some languages use the *resumption model of exception handling* in which, after handling the exception, control returns to the point at which the exception was raised and execution resumes from that point.

```
1  # Fig. 12.1: fig12_01.py
2  # Simple exception handling example.
3
4  number1 = raw_input( "Enter numerator: " )
5  number2 = raw_input( "Enter denominator: " )
6
7  # attempt to convert and divide values
8  try:
9     number1 = float( number1 )
10    number2 = float( number2 )
11    result = number1 / number2
12
13 # float raises a ValueError exception
14 except ValueError:
15    print "You must enter two numbers"
16
17 # division by zero raises a ZeroDivisionError exception
18 except ZeroDivisionError:
19    print "Attempted to divide by zero"
20
21 # else clause's suite executes if try suite raises no exceptions
22 else:
23    print "%.3f / %.3f = %.3f" % ( number1, number2, result )
```

```
Enter numerator: 100
Enter denominator: 7
100.000 / 7.000 = 14.286
```

```
Enter numerator: 100
Enter denominator: hello
You must enter two numbers
```

```
Enter numerator: 100
Enter denominator: 0
Attempted to divide by zero
```

Fig. 12.1 Exception handling with **try**, **except** and **else**.

denominator. The program then attempts to convert the user-entered values to floating-point values and to divide the numerator by the denominator. Lines 8–11 begin a **try** statement enclosing the code that may raise exceptions. Notice that the code in the **try** suite does not itself contain any **raise** statements and therefore may not appear to raise exceptions. In general, the statements in a **try** suite may call other code that possibly raises exceptions; or the statements in a **try** suite may raise exceptions if, for example, the code accesses an invalid sequence subscript, dictionary key or object attribute. In Fig. 12.1, the **try** suite makes two calls to function **float** (that may raise a **ValueError** exception) and performs one division operation (that may raise a **ZeroDivisionError** exception).

Software Engineering Observation 12.5

Place in a `try` suite a significant logical section of program in which several statements can raise exceptions, rather than using a separate `try` statements for every statement that raises an exception. However, for proper exception-handling granularity, each `try` statement should enclose a section of code small enough that, when an exception occurs, the specific context is known and the `except` handlers can process the exception properly. If many statements in a `try` suite raise the same exception types, multiple `try` statements may be required to determine each exception's context.

Function `float` converts the user-entered values to floating-point values (lines 9–10). This function raises a `ValueError` exception if it cannot convert its string argument to a floating-point value. If lines 9–10 properly convert the values (i.e., no exceptions occur), then line 11 divides the numerator by the denominator and assigns the result to variable `result`. If the denominator is zero, line 11 causes the Python interpreter to raise a `ZeroDivisionError` exception. If line 11 does not cause an exception, then the `try` suite completes its execution. If no exceptions occur in the `try` suite, the program ignores the `except` handlers in lines 14–15 and 18–19 and continues program execution with the first statement of the `else` suite (lines 22-23). The `else` suite contains a single line that prints the result of division. After the `else` suite terminates, program execution continues with the first statement after the entire `try` statement (i.e., after line 23). In this example, the program contains no more statements, so program execution terminates.

Common Programming Error 12.3

It is a syntax error to place statements between a `try` suite and its first `except` handler, between `except` handlers, between the last `except` handler and the `else` clause, or between the `try` suite and the `finally` clause.

Testing and Debugging Tip 12.4

Although a `try` suite can contain any type of statement, generally a `try` suite should contain only statements that may raise exceptions. Place in an `else` suite could that does not raise exceptions and should execute only if no exceptions occur in the corresponding `try` suite.

Immediately following the `try` suite are two `except` clauses (also called `except` handlers or exception handlers)—lines 14–15 define the exception handler for a `ValueError` exception and lines 18–19 define the exception handler for the `ZeroDivisionError` exception. Each `except` clause begins with keyword `except` followed by an exception name that specifies the type of exception handled by the `except` clause, followed by a colon (`:`). The exception-handling code appears in the body of the `except` clause (i.e., in the indented code suite). In general, when an exception occurs in a `try` suite, an `except` clause catches the exception and handles it. In Fig. 12.1, the first `except` clause specifies that it catches `ValueError` exceptions (raised by function `float`). The second `except` clause specifies that it catches `ZeroDivisionError` exceptions (raised by the interpreter). Only the matching `except` handler executes if an exception occurs. Both the exception handlers in this example display an error message. When program control reaches the last statement of an `except` handler's suite, the interpreter considers the exception handled, and program control continues with the first statement after the entire `try` statement (the end of the program in this example).

Testing and Debugging Tip 12.5

An `except` handler always should specify the class name(s) of the exception(s) to catch. An empty `except` handler should be used only for a default catch-all case.

In the second input/output dialog, the user input the string `"hello"` as the denominator. When line 10 executes, `float` cannot convert this string value to a floating-point value, so `float` raises a `ValueError` exception to indicate that the function was unable to perform the conversion. When an exception occurs, the `try` suite expires (terminates) immediately. Next, the interpreter attempts to locate a matching `except` handler starting with the `except` at line 14. The interpreter compares the type of the raised exception (`ValueError`) with the type following keyword `except` (also `ValueError`). A match occurs, so that exception handler executes, and the interpreter ignores all other exception handlers following the corresponding `try` suite. If a match did not occur, the interpreter compares the type of the raised exception with the next `except` handler in sequence and repeats the process until a match is found.

Software Engineering Observation 12.6

An `except` clause can specify more than one exception with a comma-separated sequence of exception names in parentheses, following keyword `except`. If an `except` clause specifies more than one exception, the exceptions should be related in some way (e.g., the exceptions all are caused by mathematical errors). Use a separate `except` clause for each group of related exceptions.

In the third input/output dialog of Fig. 12.1, the user input `0` as the denominator. When line 11 executes, the interpreter raises a `ZeroDivisionError` exception to indicate an attempt to divide by zero. Once again, the `try` suite terminates immediately upon encountering the exception and the interpreter attempts to locate a matching `except` handler, starting from the `except` handler at line 14. The interpreter compares the type of the raised exception (`ZeroDivisionError`) with the type following keyword `except` (`ValueError`). In this case, there is no match, because `ZeroDivisionError` and `ValueError` are not the same exception types and `ValueError` is not a base class of `ZeroDivisionError`. So, the interpreter proceeds to line 18 and compares the type of the raised exception (`ZeroDivisionError`) with the type following keyword `except` (`ZeroDivisionError`). A match occurs, so exception handler executes. If there were additional `except` handlers, the interpreter would ignore them.

12.5 Python `Exception` Hierarchy

This section overviews several of Python's exception classes. All exceptions inherit from base class ***`Exception`*** and are defined in module ***`exceptions`***. Python automatically places all exception names in the built-in namespace, so programs do not need to import the `exceptions` module to use exceptions. Python defines four primary classes that inherit from `Exception`—***`SystemExit`***, ***`StopIteration`***, ***`Warning`*** and ***`StandardError`***. Exception `SystemExit`, when raised and left uncaught, terminates program execution. If an interactive session encounters an uncaught `SystemExit` exception, the interactive session terminates. Python uses exception `StopIteration` (new in version 2.2) to determine when a `for` loop reaches the end of its sequence. Python uses `Warning` exceptions to indicate that certain elements of Python may change in the future.

For example, if a Python 2.2 program uses a variable named **yield**, Python raises a **Warning** exception, because future versions of Python, will reserve **yield** for use as a keyword. **StandardError** is the base class for all Python error exceptions (e.g., **ValueError** and **ZeroDivisionError**).

Figure 12.2 contains the exception hierarchy for Python 2.2. For any version of Python, the programmer can obtain the exception hierarchy with the statements

```
import exceptions
print exceptions.__doc__
```

Many **StandardError** exceptions can be caught at runtime and handled, so the program can continue running. Such exceptions often can be avoided by coding properly. For example, if a program attempts to access an out-of-range sequence subscript, the interpreter raises an exception of type ***IndexError***. Similarly, an ***AttributeError*** exception occurs when a program attempts to access a non-existent object attribute.

One of the benefits of the exception class hierarchy is that an **except** handler can catch exceptions of a particular type or can use a base-class type to catch exceptions in a hierarchy of related exception types. For example, Section 12.3 discussed the empty **except** handler, which catches exceptions of all types. An **except** handler that specifies an exception of type **Exception** also can catch all exceptions (assuming the raised exceptions inherit from class **Exception**), because **Exception** is the base class of all exception classes.

Using inheritance with exceptions enables an exception handler to catch related exceptions with a concise notation. An exception handler certainly could catch each derived-class exception individually, but it is more concise to catch the base-class exception if the handling behavior is the same for all derived classes. Otherwise, catch each derived-class exception individually.

Common Programming Error 12.4

It is a syntax error to place an empty ***except*** *clause before the last* ***except*** *clause following a particular* ***try*** *suite.*

Common Programming Error 12.5

It is a logic error if two or more ***except*** *clauses following a particular* ***try*** *suite specify the exact same exception type. Python executes the first* ***except*** *handler that matches a raised exception and ignores any additional* ***except*** *handlers that catch the same exception type.*

Common Programming Error 12.6

Placing an ***except*** *handler that catches type* ***Exception*** *before other* ***except*** *handlers is a logic error, because all exceptions would be caught before other exception handlers could be reached. Thus, subsequent exception handlers are unreachable.*

Determining when Python and standard and third-party components raise exceptions can be difficult—there is no way for a program to determine whether, for example, a function may raise a particular exception. The language reference and standard library documentation[3] often specify cases in which exceptions are raised. For example, in Fig. 12.1,

3. The library reference can be found at **www.python.org/doc/current/lib/lib.html**, and the language reference can be found at **www.python.org/doc/current/ref/ref.html**.

Python exceptions

```
Exception
   SystemExit
   StopIteration
   StandardError
      KeyboardInterrupt
      ImportError
      EnvironmentError
         IOError
         OSError
            WindowsError  (Note: Defined on Windows platforms only)
      EOFError
      RuntimeError
         NotImplementedError
      NameError
         UnboundLocalError
      AttributeError
      SyntaxError
         IndentationError
            TabError
      TypeError
      AssertionError
      LookupError
         IndexError
         KeyError
      ArithmeticError
         OverflowError
         ZeroDivisionError
         FloatingPointError
      ValueError
         UnicodeError
      ReferenceError
      SystemError
      MemoryError
   Warning
      UserWarning
      DeprecationWarning
      SyntaxWarning
      OverflowWarning
      RuntimeWarning
```

Fig. 12.2 Python exception hierarchy.

we demonstrated that Python raises a **`ZeroDivisionError`** exception when a program attempts to divide by zero. In the language reference, Section 5.6 discusses the division

operator and states that division by zero causes a **ZeroDivisionError** exception. A third-party component intended for distribution and use in software development also should include documentation that indicates the exceptions raised by the component and why such exceptions occur.

Software Engineering Observation 12.7

If a component raises exceptions, the component documentation should state that the component raises the exception. Statements that use the component should be placed in **try** *suites, and those exceptions should be caught and handled.*

12.6 `finally` Clause

Programs frequently request and release resources dynamically (i.e., at execution time). For example, a program that reads a file from disk first asks to open that file. If that request succeeds, the program reads the contents of the file. Operating systems typically can prevent more than one program from manipulating a file at once. Therefore, when a program finishes processing a file, the program normally closes the file (i.e., releases the resource). This enables other programs to use the file. Closing the file helps prevent a *resource leak*, in which the file resource is not available to other programs because a program using the file never closed it. Programs that obtain certain types of resources (such as files) typically should return those resources explicitly to the system to avoid resource leaks.

In programming languages (e.g., C and C++) in which programmers are responsible for dynamic memory management, the most common type of resource leak is a *memory leak*. This happens when a program allocates (obtains) memory, but does not deallocate (release) the memory when it is no longer needed. In Python, normally this is not an issue, because the interpreter performs *garbage collection* of memory no longer needed by an executing program. However, other kinds of resource leaks (such as the unclosed files mentioned previously) can occur in Python.

Testing and Debugging Tip 12.6

The interpreter does not eliminate memory leaks completely. The interpreter will not garbage-collect an object while references to that object exist. Thus, memory leaks can occur if programmers erroneously keep references to unwanted objects.

Most resources that require explicit release have potential exceptions associated with processing those resources. For example, a program that processes a file might receive **IOError** exceptions during the processing. For this reason, file processing code normally appears in a **try** suite. Regardless of whether a program successfully processes a file, the program should close the file when the file is no longer needed.

Suppose a program places all resource-request and resource-release code in a **try** suite. If no exceptions occur, the **try** suite executes normally and releases the resources. However, if an exception occurs, the **try** suite expires before the resource-release code can execute. We could duplicate all resource-release code in the **except** handlers, but this makes the code more difficult to modify and maintain.

Python's exception handling mechanism provides the ***finally*** clause, which is guaranteed to execute if program control enters the corresponding **try** suite, regardless of whether that **try** suite executes successfully or an exception occurs. This guarantee makes the **finally** suite an ideal location to place resource-deallocation code for resources acquired and manipulated in the corresponding **try** suite. If the **try** suite executes suc-

cessfully, the **finally** suite executes immediately after the **try** suite terminates. If an exception occurs in the **try** suite, the **finally** suite executes immediately after the line that caused the exception. The exception is then processed by the next enclosing **try** statement (if there is one).

Testing and Debugging Tip 12.7

*A **finally** suite typically contains code to release resources acquired in the corresponding **try** suite, making the **finally** suite an effective way to eliminate resource leaks.*

Testing and Debugging Tip 12.8

*The only reason a **finally** suite will not execute if program control entered the corresponding **try** suite is if the application terminates before the **finally** can execute.*

Performance Tip 12.2

As a rule, resources should be released as soon as they are no longer needed in a program. This makes those resources available for reuse immediately and enables other programs to access those resources.

Software Engineering Observation 12.8

Before raising an exception, the code that raises the exception should release any resources acquired in the code before the exception occurred.

Figure 12.3 demonstrates that the **finally** clause always executes, regardless of whether an exception occurs in the corresponding **try** suite. The program consists of two functions to demonstrate **finally**—**doNotRaiseException** (lines 4–14) and **raiseExceptionDoNotCatch** (lines 16–27). The main program calls these functions to demonstrate when **finally** clauses execute.

```
1  # Fig. 12.3: fig12_03.py
2  # Using finally clauses.
3
4  def doNotRaiseException():
5
6     # try block does not raise any exceptions
7     try:
8        print "In doNotRaiseException"
9
10    # finally executes because corresponding try executed
11    finally:
12       print "Finally executed in doNotRaiseException"
13
14    print "End of doNotRaiseException"
15
16 def raiseExceptionDoNotCatch():
17
18    # raise exception, but do not catch it
19    try:
20       print "In raiseExceptionDoNotCatch"
21       raise Exception
22
```

Fig. 12.3 **finally** always executes. (Part 1 of 2.)

```
23    # finally executes because corresponding try executed
24    finally:
25       print "Finally executed in raiseExceptionDoNotCatch"
26
27    print "Will never reach this point"
28
29 # main program
30
31 # Case 1: No exceptions occur in called function.
32 print "Calling doNotRaiseException"
33 doNotRaiseException()
34
35 # Case 2: Exception occurs, but is not handled in called function,
36 # because no except clauses exist in raiseExceptionDoNotCatch
37 print "\nCalling raiseExceptionDoNotCatch"
38
39 # call raiseExceptionDoNotCatch
40 try:
41    raiseExceptionDoNotCatch()
42
43 # catch exception from raiseExceptionDoNotCatch
44 except Exception:
45    print "Caught exception from raiseExceptionDoNotCatch " + \
46       "in main program."
```

```
Calling doNotRaiseException
In doNotRaiseException
Finally executed in doNotRaiseException
End of doNotRaiseException

Calling raiseExceptionDoNotCatch
In raiseExceptionDoNotCatch
Finally executed in raiseExceptionDoNotCatch
Caught exception from raiseExceptionDoNotCatch in main program.
```

Fig. 12.3 **`finally`** always executes. (Part 2 of 2.)

Line 33 of the main program calls function **`doNotRaiseException`** (lines 4–14)—a function that contains a **`try`**/**`finally`** form. The **`try`** suite (line 8) outputs a message. The **`try`** suite does not raise any exceptions, so program control reaches the end of the suite. Next, the **`finally`** clause's suite (line 12) executes and outputs a message. At this point, program control continues with the first statement after the **`finally`** suite, because no exception was raised. This statement (line 14) outputs a message indicating that the end of the function has been reached. Then, program control returns to the main program.

Common Programming Error 12.7

It is a syntax error to write a **`try`** *statement that does not contain either a* **`finally`** *clause or one or more* **`except`** *clauses. If a* **`try`** *statement does not have any* **`except`** *clauses, it must have a* **`finally`** *clause. If a* **`try`** *statement does not have a* **`finally`** *clause, it must have one or more* **`except`** *clauses.*

Lines 40–41 of the main program begin a **`try`** statement that invokes function **`raise-ExceptionDoNotCatch`** (lines 16–27). The **`try`** statement enables the main program to

catch any exceptions raised by **raiseExceptionDoNotCatch**. In **raiseExceptionDoNotCatch**, the **try** suite (lines 20–21) begins by outputting a message. Next, the **try** suite raises an **Exception** (line 21) and the **try** suite expires immediately. This **try** statement does not specify any **except** clauses; therefore, the exception is not caught in function **raiseExceptionDoNotCatch**. Normal program control cannot continue until that exception is caught and processed. Thus, the interpreter will terminate **raiseExceptionDoNotCatch** and program control will return to the main program. Before control returns to the main program, however, the **finally** clause's suite (line 25) executes and outputs a message. At this point, program control returns to the main program—any statements appearing after the **finally** suite (e.g., line 27) do not execute. In the main program, the **except** handler in lines 44–46 catches the exception and displays a message indicating that the exception was caught in the main program.

Common Programming Error 12.8

*Raising an exception in a **finally** suite is a potentially dangerous operation. If an uncaught exception is awaiting processing when the **finally** suite executes and the **finally** suite raises a new exception that the suite does not catch, the first exception is lost, and the new exception is passed to the next enclosing **try** statement.*

Testing and Debugging Tip 12.9

*In a **finally** suite, always enclose in a **try** statement code that may raise an exception. This prevents losing uncaught exceptions that occur before the **finally** suite executes.*

Software Engineering Observation 12.9

*If a **try** statement specifies a **finally** clause, the **finally** clause's suite executes even if the **try** suite is terminated by a **return** statement. Then, the **return** to the calling code occurs.*

Note that the point at which program control continues after the **finally** clause executes depends on the exception-handling state. If the **try** suite successfully completes, the **finally** suite executes and control continues with the next statement after the **finally** suite. If the **try** suite raises an exception, the **finally** suite executes then program control continues in the next enclosing **try** statement. The enclosing **try** may be in the calling function or one of its callers. It also is possible to nest a **try**/**except** form in a **try** suite, in which case the outer **try** statement's exception handlers would process any exceptions the were not caught in the inner **try** statement.

12.7 Exception Objects and Tracebacks

As we discussed in Section 12.5, exception data types—which derive from class **Exception**—can be created with zero or more arguments. These arguments frequently are used to formulate error messages for a raised exception. When Python creates an exception object in response to a **raise** statement, Python places any arguments from the **raise** statement in the exception object's ***args*** attribute.

When an exception occurs, Python "remembers" the exception that has been raised and the current state of the program. Python also maintains ***traceback*** objects that contain information about the function call stack from the time the exception occurred. Recall that exceptions can be raised in a deeply nested series of function calls. As the program calls

each function, Python inserts the function name at the beginning of the *function call stack*. When an exception is raised, Python begins searching for an exception handler. If no exception handler exists in the current function, the current function terminates execution, and Python searches the current function's calling function, and so on, until either an exception handler is found or Python reaches the main program. This process of searching for an appropriate exception handler is called *stack unwinding*. Just as the interpreter maintains information about functions that are placed on the stack, the interpreter maintains information about functions that have been unwound from the stack.

Testing and Debugging Tip 12.10

A traceback shows the complete function call stack from the time at which an exception occurred. This lets the programmer view the series of function calls that led to the exception. Information in the traceback includes names of unwound functions, names of the files in which the functions are defined and line numbers that indicate where the program encountered an error. The last line number in the traceback indicates the throw point (i.e., the location where the original exception was raised). Previous line numbers indicate the locations from which each function in the traceback was called.

Our next example (Fig. 12.4) demonstrates exception object's **`args`** attribute and exception object string representation. The example also demonstrates how to access **`traceback`** objects to print information about stack unwinding. As we discuss this example, we keep track of the functions on the call stack so we can discuss the **`traceback`** object and the stack-unwinding mechanism.

```
# Fig. 12.4: fig12_04.py
# Demonstrating exception arguments and stack unwinding.

import traceback

def function1():
   function2()

def function2():
   function3()

def function3():

   # raise exception, catch exception, reraise exception
   try:
      raise Exception, "An exception has occurred"
   except Exception:
      print "Caught exception in function3. Reraising....\n"
      raise   # reraises most recent exception

# call function1, any Exception it generates will be
# caught by the except clause that follows
try:
   function1()

```

Fig. 12.4 Exception arguments and stack unwinding. (Part 1 of 2.)

```
26 # output exception arguments, string representation of exception,
27 # and the traceback
28 except Exception, exception:
29    print "Exception caught in main program."
30    print "\nException arguments:", exception.args
31    print "\nException message:", exception
32    print "\nTraceback:"
33    traceback.print_exc()
```

```
Caught exception in function3. Reraising....

Exception caught in main program.

Exception arguments: ('An exception has occurred',)

Exception message: An exception has occurred

Traceback:
Traceback (most recent call last):
  File "fig12_04.py", line 24, in ?
    function1()
  File "fig12_04.py", line 7, in function1
    function2()
  File "fig12_04.py", line 10, in function2
    function3()
  File "fig12_04.py", line 16, in function3
    raise Exception, "An exception has occurred"
Exception: An exception has occurred
```

Fig. 12.4 Exception arguments and stack unwinding. (Part 2 of 2.)

The interpreter begins executing the program with line 1. This is technically the first line in the main program. The main program is the first entry in the function call stack, because it is the entity that invokes all other functions. Line 24 of the **`try`** suite in the main program invokes **`function1`** (defined in lines 6–7), which becomes the second entry on the stack. If **`function1`** raises an exception, the **`except`** handler in lines 28–33 catch the exception and output information about the exception that occurred. Line 7 of **`function1`** invokes **`function2`** (defined in lines 9–10), which becomes the third entry on the stack. Then, line 10 of **`function2`** invokes **`function3`** (defined in lines 12–19) which becomes the fourth entry on the stack.

At this point, the call stack for the program is

`function3` *(top)*
`function2`
`function1`
Main Program

with the last function called (**`function3`**) at the top and the main program at the bottom. Line 16 in **`function3`** raises an **`Exception`** and passes the string **`"An exception has occurred"`** as an argument. In response to the **`raise`** statement, Python creates an **`Exception`** object, with the specified argument. The **`except`** clause in lines 17–19 catches the exception and first prints a message. Line 19 uses an empty **`raise`** statement

to *reraise* the exception. Usually, reraising an exception indicates that the **except** handler performed partial processing of the exception and is now passing the exception back to the caller (in this case **function2**) for further processing. In this example, the **function3** demonstrates that keyword **raise**, with no specified exception name, reraises the most recently raised exception.

Software Engineering Observation 12.10

If a function is capable of handling a given type of exception, then let that function handle it, rather than passing the exception to another region of the program.

Next, **function3** terminates because the reraised exception is not caught in the function body. Thus, control will return to the statement that invoked **function3** in the prior function in the call stack (**function2**). This removes or *unwinds* **function3** from the function call stack (thus, terminating the function) and Python maintains information about the function call in a **traceback** object.

When control returns to line 10 in **function2**, the interpreter ascertains that line 10 is not in a **try** suite. Therefore, the exception cannot be caught in **function2**, and **function2** terminates. This unwinds **function2** from the function call stack, creates another **traceback** object (to represent the current level of unwinding) and returns control to line 7 in **function1**. Here again, line 7 is not in a **try** suite, so the exception cannot be caught in **function1**. The function terminates and unwinds from the call stack, creating another **traceback** object and returning control to line 24 in the main program, which is in a **try** suite. The **try** suite in the main program expires and the **except** handler in lines (28–33) catches the exception.

Notice that the **except** clause in line 28 differs from the **except** clauses presented thus far. When Python encounters an **except** clause in which **except** is followed by an exception type (or tuple of exception types), a comma, and an identifier, Python binds the identifier to the matching exception object. Now, the **except** handler can use the identifier to obtain information about the specific exception that occurred. The **except** suite in lines 29–33 prints the exception object's **args** attribute (line 30). Then, the handler prints the string representation of the exception. Python's string representation of an exception object depends on the value of its **args** attribute. If the **args** attribute is an empty tuple, Python represents the exception as the empty string. If an exception objects's **args** tuple contains only one value, Python's represents the exception as the string representation of that value. If an exception object's **args** tuple contains multiple items, Python represents the exception as the string representation of the **args** tuple. In this example, the exception object's **arg** attribute contains only one value, so Python represents the exception as that value (i.e., the string **"An exception has occurred"**).

Line 33 of the **except** handler calls function ***traceback.print_exc*** to print the traceback. Module **traceback** contains many functions for manipulating the **traceback** objects that Python creates during stack unwinding. Recall that stack unwinding continues until either an **except** handler catches the exception or the program terminates. Function **print_exc**, when called with no arguments, prints all the **traceback** objects accumulated thus far in the stack-unwinding process. This output is identical to the output Python produces when the interpreter encounters an uncaught exception. Let us examine the output from function **print_exc**. The first line

```
Traceback (most recent call last)
```

is the standard traceback line that Python prints when an error occurs. This line indicates that the most recent call (i.e., the call at the top of the call stack when the exception occurred) appears last in the traceback output. The next two lines in the traceback output contain information about the first call on the function call stack (i.e., the call to **`function1`** from the main program). The information includes the file in which the call occurred (**`fig12_04.py`**), the line number of the file that called the function (**`24`**) and the calling entity from which the function was invoked (**`?`**, which corresponds to the main program). The subsequent pairs of lines in the traceback output each correspond to a call on the function call stack. The second-to-last line contains the code that caused the exception (i.e., the code from line 16 in **`function3`** that contains the **`raise`** statement). This demonstrates the fact that the empty **`raise`** statement in line 19 simply reraises the exception from line 16. The final line of the output contains a string representation of the exception type and its argument. Note that traceback output contains information about the call stack from the point at which the exception occurred to the point at which the exception is caught (or the point at which the program terminates, if the exception is not caught).

Testing and Debugging Tip 12.11

When reading a traceback, start from the end of the traceback and read the error message first. Then, read up the remainder of the traceback, looking for the first line that indicates code that you wrote in your program. Normally, this is the location that caused the exception.

12.8 Programmer-Defined Exception Classes

In many cases, programmers can use existing exception classes from the Python hierarchy to indicate exceptions that occur in their programs. However, in some cases, programmers may wish to create new exception types that are more specific to the problems that occur in their programs. *Programmer-defined exception classes* should derive directly or indirectly from class **`Exception`**.

Good Programming Practice 12.1

Associating each type of malfunction with an appropriately named exception class improves program clarity.

Good Programming Practice 12.2

Before creating programmer-defined exception classes, investigate the existing exception classes in the Python hierarchy to discover whether an appropriate exception type already exists.

Good Programming Practice 12.3

Define new exception classes only if programmers need to catch and handle the new exceptions differently from other existing exception types.

Figure 12.5 demonstrates defining and using a programmer-defined exception class. Class **`NegativeNumberError`** (lines 6–8) is a programmer-defined exception class representing exceptions that occur when a program performs an illegal operation on a negative number, such as the square root of a negative number.

Lines 6–8 define a programmer-defined exception class. The Python exception class hierarchy defines many categories of exceptions, and programmer-defined exceptions should extend an appropriate exception from one of these categories. **`NegativeNumberError`**

```python
# Fig. 12.5: fig12_05.py
# Demonstrating a programmer-defined exception class.

import math

class NegativeNumberError( ArithmeticError ):
   """Attempted improper operation on negative number."""
   pass

def squareRoot( number ):
   """Computes square root of number. Raises NegativeNumberError
   if number is less than 0."""

   if number < 0:
      raise NegativeNumberError, \
         "Square root of negative number not permitted"

   return math.sqrt( number )

while 1:

   # get user-entered number and compute square root
   try:
      userValue = float( raw_input( "\nPlease enter a number: " ) )
      print squareRoot( userValue )

   # float raises ValueError if input is not numerical
   except ValueError:
      print "The entered value is not a number"

   # squareRoot raises NegativeNumberError if number is negative
   except NegativeNumberError, exception:
      print exception

   # successful execution: terminate while loop
   else:
      break
```

```
Please enter a number: hello
The entered value is not a number

Please enter a number: -900
Square root of negative number not permitted

Please enter a number: 12.345
3.51354521815
```

Fig. 12.5 Programmer-defined exception class.

exceptions most likely occur during arithmetic, so it seems logical to derive class **`NegativeNumberError`** from class **`ArithmeticError`**. Creating simple, programmer-defined exceptions in Python is easy, because the new exception class inherits all its functionality

from the base-class exception. Therefore, the body of the class contains only the keyword **pass**—the keyword that indicates a suite or block performs no work.

The remainder of the program (lines 10–37) demonstrates our programmer-defined exception class. The program enables the user to input a numeric value, then invokes function **squareRoot** (lines 10–18) to calculate the square root of that value. For this purpose, **squareRoot** invokes function **math.sqrt**, which wants a nonnegative value as its argument. If **math.sqrt** receives a negative value, the function raises a **ValueError** exception with the argument **"math domain error"**. In this program, we essentially write our own square root function that uses a programmer-defined exception to prevent the user from calculating the square root of a negative number. If the numeric value received from the user is negative, function **squareRoot** raises a **NegativeNumberError** (lines 14–16). Otherwise, **squareRoot** invokes function **math.sqrt** to compute the square root.

In the main program, a **while** loop (lines 20–37) continues executing until the user enters a nonnegative value. The **try** suite (lines 24–25) attempts to obtain a numerical value from the user and to pass that value to function **squareRoot**. When the user inputs a value and presses *Enter*, the program passes the user-entered value to function **float**. If the value is not a number, function **float** raises a **ValueError** exception, and the except handler in lines 28–29 prints an error message. Control then returns to the beginning of the **while** loop. If the user inputs a negative number, function **squareRoot** raises a **NegativeNumberError**. The except handler in lines 32–33 simply prints the exception object before control returns to the beginning of the **while** loop. If the user enters a valid, nonnegative number, line 25 prints the square root of the number before program control proceeds to the **else** clause in lines 36–37. The **else** suite contains only the keyword **break**, which terminates the **while** loop.

In this chapter, we demonstrated how the exception-handling mechanism works and discussed how to make applications more robust by writing exception handlers to process potential problems. When developing new applications, it is important to investigate potential exceptions raised by the functions your program invokes or by the interpreter, then implement appropriate exception-handling code to make those applications more robust. In Chapter 13, String Manipulation and Regular Expressions, we begin a discussing a series of techniques for developing substantial software. These techniques, when combined with disciplined exception handling, enable Python programmers to create viable, valuable software components.

SUMMARY

- An exception is an indication of a “special event” that occurs during a program’s execution. Often the special event is an error (e.g., dividing by zero or adding two incompatible types). Sometimes the special event is something else (e.g., the termination of a **for** loop).
- Exception handling enables programmers to write clear, robust, more fault-tolerant programs that can resolve (or handle) exceptions.
- The style and details of exception handling in Python are based on the Modula-3 language. This exception-handling mechanism is similar to that used in C# and Java.
- The **raise** statement executes to indicate that an exception has occurred. This is called raising (or sometimes throwing) an exception.
- The simplest **raise** statement consists of the keyword **raise**, followed by the name of the exception to be raised.

• Exception names specify classes and Python exceptions are objects of those classes. When the **raise** statement executes, Python creates an object of the specified exception class.
• The **raise** statement may specify an argument or arguments that initialize the exception object. In this case, a comma follows the exception name, and the argument or a tuple of arguments follows the comma.
• Exception handling enables the programmer to remove error-handling code from the "main line" of the program's execution. This improves program clarity and enhances modifiability.
• Programmers can decide to handle whatever exceptions they choose—all types of exceptions, all exceptions of a certain type or all exceptions of related types.
• The exception-handling mechanism is useful for processing problems that occur when a program interacts with reusable software components. Rather than internally handling problems that occur, such components use exceptions to notify client code of problems. This enables programmers to implement error handling that is appropriate to each application.
• Exception handling is geared to situations in which the code that detects an error is unable to handle it. Such code raises or throws an exception.
• Python uses **try** statements to enable exception handling. The **try** statement encloses statements that potentially cause exceptions. A **try** statement consists of keyword **try**, followed by a colon (**:**), followed by a suite of code in which exceptions may occur, followed by one or more clauses.
• Immediately following the **try** suite may be one or more **except** clauses (also called **except** handlers). Each **except** clause specifies zero or more exception names that represent the type(s) of exceptions the **except** clause can handle.
• The **except** clause also may specify an identifier for the exception that was raised, and the handler can use the exception object to obtain information about that exception.
• An **except** clause that specifies no exception type is an empty **except** clause, which catches all exception types. It is a syntax error to place an empty **except** clause before any other **except** clauses in a particular **try** statement.
• After the last **except** clause, an optional **else** clause contains code that executes if the code in the **try** suite raised no exceptions.
• A **try** suite can be followed by zero **except** clauses; in that case, it must be followed by a **finally** clause. The code in the **finally** suite always executes, regardless of whether an exception occurs.
• Programmers sometimes refer to the point in the program at which an exception occurs as the throw point.
• Exceptions are objects of classes that inherit from class **Exception**.
• If an exception occurs in a **try** suite, the **try** suite expires and program control transfers to the first matching **except** handler (if there is one) following the **try** suite. A match occurs if the types are identical or if the raised exception's type is a derived class of the handler's exception type.
• If no exceptions occur in a **try** suite, the interpreter ignores the exception handlers for that **try** statement.
• If an exception occurs in a statement that is not in a **try** suite and that statement is in a function, the function containing that statement terminates immediately and the interpreter attempts to locate an enclosing **try** statement in a calling function—a process called stack unwinding.
• Python is said to use the termination model of exception handling, because the **try** statement enclosing a raised exception expires immediately when that exception occurs.
• Function **float** raises a **ValueError** exception if the function cannot convert its argument value to a floating-point value.

- The Python interpreter automatically tests for division by zero and raises a **`ZeroDivisionError`** exception if the denominator is zero.
- As good programming practice, an **`except`** handler always should specify the name of the exception to catch. An empty **`except`** handler should be used only for a default catch-all case.
- The preferred exception-handling mechanism is to allow objects of class **`Exception`** and its derived classes to be raised and caught.
- An **`except`** handler can catch exceptions of a particular type or can use a base-class type to catch exceptions in a hierarchy of related exception types.
- A third-party component intended for distribution and use in software development also should include documentation that indicates which exceptions are raised by the component.
- Programs frequently request and release resources dynamically. Programs that obtain certain types of resources (such as files) sometimes must return those resources explicitly to the system to avoid resource leaks. Most resources that require explicit release have potential exceptions associated with processing those resources.
- The **`finally`** clause that guaranteed to execute if program control enters the corresponding **`try`** suite. The **`finally`** clause is an ideal location to place resource deallocation code for resources acquired and manipulated in the corresponding **`try`** suite.
- Objects of exception data types can be created with zero or more arguments. These arguments frequently are used to formulate error messages for a raised exception.
- When Python creates an exception object in a **`raise`** statement, Python places any arguments from the **`raise`** statement in the exception object's **`args`** attribute.
- When an exception occurs, Python remembers the exception that was raised and the current state of the program. Python also maintains **`traceback`** objects that contains information about the function call stack from the time the exception occurred.
- Python maintains information about functions that have been unwound from the stack with **`traceback`** objects.
- An empty **`raise`** statement reraises the most recently raised exception.
- When Python encounters an **`except`** clause in which **`except`** is followed by an exception type (or tuple of exception types), a comma and an identifier, Python binds the identifier to the exception object that the except handler catches.
- If an exception object's **`args`** attribute is an empty tuple, the exception's string representation is the empty string.
- If an exception objects's **`args`** tuple contains only one value, the exception's string representation is the string representation of that value.
- If an exception object's **`args`** tuple contains multiple items, the exception's string representation is the string representation of the **`args`** tuple.
- Module **`traceback`** contains many functions for manipulating the **`traceback`** objects that Python creates during stack unwinding.
- Function **`traceback.print_exc`**, when called with no arguments, prints all the **`traceback`** objects accumulated thus far in the stack-unwinding process.
- A Python **`traceback`** object stores information about a function call, including the file name, line numbers and the code that caused an error.
- Programmer-defined exception classes should derive directly or indirectly from class **`Exception`**.
- If a programmer-defined exception requires no extra functionality, the programmer can create the exception merely by inheriting from an existing exception class and placing keyword **`pass`** in the body of the class.

TERMINOLOGY

args attribute of exception object
automatic garbage collection
call stack
catch related errors
divide by zero
eliminate resource leaks
empty **except** clause
empty **raise** clause
error-processing code
except handler
except clause
except suite expires
exception
Exception class
exception handler
fault-tolerant program
finally clause
FormatException class
function call stack
garbage collection
IndexError exception
inheritance with exceptions
memory exhaustion
memory leak
Modula-3
out-of-range sequence subscript
print_exc function of module **traceback**
raise an exception
raise statement
release a resource
reraise an exception
resource leak
resumption model of exception handling
sqrt function of module **math**
stack unwinding
StandardError exception
StopIteration exception
SystemExit exception
termination model of exception handling
throw an exception
throw point
traceback module
traceback object
try statement
try/**except** form
try/**except**/**else** form of a **try** statement
try/**finally** form of a **try** statement
programmer-defined exception class
Warning exception
ZeroDivisionError exception

SELF-REVIEW EXERCISES

12.1 Fill in the blanks in each of the following statements:

a) Python uses exception handling to determine when a __________ loop terminates.
b) A function is said to __________ an exception when it detects that a problem occurred.
c) The __________ clause, if it appears after a **try** suite, always executes.
d) Most basic Python exceptions derive from class __________.
e) The statement that raises an exception is sometimes called the __________ of the exception.
f) A __________ statement encloses code that may raise an exception.
g) If the catch-all exception handler is specified before another exception handler, a __________ may occur.
h) An uncaught exception in a function causes that function to be __________ from the function call stack.
i) Function **float** can raise a(n) __________ exception if its argument cannot be converted to a floating-point value.
j) Python maintains information about the functions unwound from the stack in __________ objects.

12.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Exceptions always are handled in the function that initially detects the exception.
b) Accessing a nonexistent object attribute causes an **AttributeError** exception.
c) Accessing an out-of-bounds sequence subscript causes the interpreter to raise an exception.
d) A **try** statement must contain one or more clauses.
e) If a **finally** clause appears in a function, that **finally** clause is guaranteed to execute.

f) In Python, it is possible to return to the throw point of an exception via keyword **`return`**.
g) Exceptions can be reraised.
h) Function **`math.sqrt`** raises a **`NegativeNumberError`** exception if called with a negative-integer argument.
i) Exception object attribute **`args`** contains a string that corresponds to the exception's error message.
j) Exceptions can be raised only by functions explicitly called in **`try`** statements.

ANSWERS TO SELF-REVIEW EXERCISES

12.1 a) **`for`**. b) raise (or throw). c) **`finally`**. d) **`Exception`**. e) throw point. f) **`try`**. g) syntax or logic error. h) unwound. i) **`ValueError`**. j) **`traceback`**.

12.2 a) False. Although it is possible to handle an exception in the function that originally detects the exception, often an exception is handled by a calling function on the function call stack. b) True. c) True. d) True. e) False. The **`finally`** clause will execute only if program control enters the corresponding **`try`** suite and if the **`try`** suite does not terminate the program. f) False. It is not possible to return control to the throw point of an exception in Python. g) True. h) False. Function **`math.sqrt`** raises a **`ValueError`** exception if called with a negative-integer argument. i) False. Exception object attribute **`args`** contains a tuple that corresponds to the arguments used to initialize the exception object. j) False. Exceptions can be raised by any code, regardless of whether it is called from a **`try`** statement. Also, the interpreter can raise exceptions.

EXERCISES

12.3 Use inheritance to create an exception base class and various exception-derived classes. Write a program to demonstrate that the **`except`** clause specifying the base class catches derived-class exceptions.

12.4 Write a Python program that demonstrates how various exceptions are caught with

```
except Exception, exception
```

12.5 Write a Python program that shows the importance of the order of exception handlers. Write two programs, one with the correct order of **`except`** handlers and another with an order that causes a logic error. If you attempt to catch a base-class exception type before a derived-class type, the program may produce a logic error.

12.6 Exceptions can be used to indicate problems that occur when an object is being constructed. Write a Python program that shows a constructor passing information about constructor failure to an exception handler that occurs after a **`try`** statement. The exception raised also should contain the arguments sent to the constructor.

12.7 Write a Python program that illustrates reraising an exception.

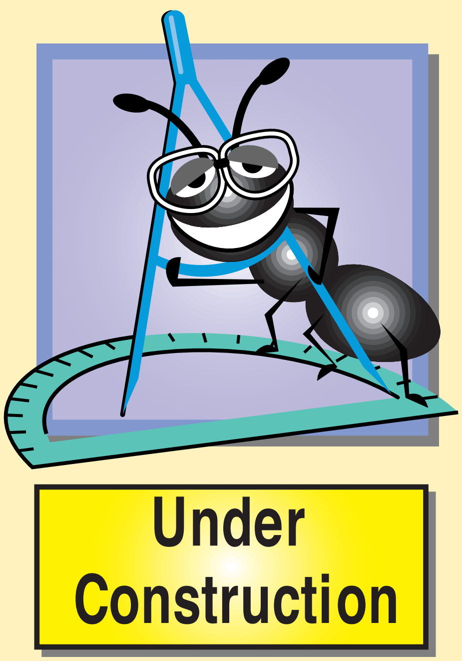
12.8 Write a Python program that shows that a function with its own **`try`** statement does not have to catch every possible exception that occurs within the **`try`** suite. Some exceptions can slip through to, and be handled in, other scopes.

13

String Manipulation and Regular Expressions

Objectives

- To understand text processing in Python.
- To use Python's string data-type methods.
- To manipulate and search string contents.
- To understand and create regular expressions.
- To use regular expressions to match patterns in strings.
- To use metacharacters, special sequences and grouping to create complex regular expressions.



The chief defect of Henry King
Was chewing little bits of string.
Hilaire Belloc

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences.
William Strunk, Jr.

I have made this letter longer than usual, because I lack the time to make it short.
Blaise Pascal

The difference between the almost-right word & the right word is really a large matter—it's the difference between the lightning bug and the lightning.
Mark Twain

Mum's the word.
Miguel de Cervantes, *Don Quixote de la Mancha*

Outline

13.1 Introduction

This chapter introduces Python's string and character processing capabilities and demonstrates using regular expressions to search for patterns in text. The techniques presented in this chapter can be employed to develop text editors, word processors, page-layout software, computerized typesetting systems and other text-processing software. Previous chapters presented several string-processing capabilities. In this chapter, we expand on this information by detailing the capabilities of various methods of the basic string data type and the powerful text-processing capabilities provided in the Python module **`re`**.

13.2 Fundamentals of Characters and Strings

Characters (digits, letters and symbols such as $, @, % and *) are the fundamental building blocks of Python programs. Every program is composed of characters that, when grouped meaningfully, represent a series of instructions that the interpreter uses to perform a task. Each character has a corresponding *character code* (sometimes called its *integer ordinal value*). For example, the integer value **`122`** corresponds to the character constant **`"z"`**. Python provides function ***`ord`*** that takes as an argument a character and returns its character code (as shown in the interactive session of Fig. 13.1). In most modern programming languages and systems, character values are established according to the *Unicode character set*—an international character set that contains many more symbols and letters than does the ASCII character set (see Appendix B, ASCII Character Set). To learn more about Unicode, see Appendix F.

Python supports strings as a basic data type. Recall that strings are immutable sequences—strings cannot be changed after they are created. We have seen how to obtain the length of a string with function **`len`**, how to concatenate strings with operator **`+`** and

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> ord( "z" )
122
>>> ord( "\n" )
10
```

Fig. 13.1 Integer ordinal value of a character.

how to format strings with format operator **%**. Strings also support methods that perform various other formatting and processing capabilities. The table in Fig. 13.2 lists the string methods. When a program invokes a string method that appears to modify the string, the method actually returns its results as a new string. In the table, the "original string" refers to the string on which a method is invoked. We discuss many of these methods in the following sections.

String Method	Description
capitalize()	Returns a version of the original string in which only the first letter is capitalized. Converts any other capital letters to lowercase.
center(*width* **)**	Returns a copy of the original string centered (using spaces) in a string of *width* characters.
count(*substring[***,** *start[***,** *end]]* **)**	Returns the number of times *substring* occurs in the original string. If argument *start* is specified, searching begins at that index. If argument *end* is indicated, searching begins at *start* and stops at *end*.
encode(*[encoding[***,** *errors]* **)**	Returns an encoded string. Python's default encoding is normally ASCII. Argument *errors* defines the type of error handling used; by default, errors is ***"strict"***.
endswith(*substring[***,** *start[***,** *end]]* **)**	Returns 1 if the string ends with *substring*. Returns 0 otherwise. If argument *start* is specified, searching begins at that index. If argument *end* is specified, the method searches through the slice *start*:*end*.
expandtabs(*[tabsize]* **)**	Returns a new string in which all tabs are replaced by spaces. Optional argument *tabsize* specifies the number of space characters that replace a tab character. The default value is 8.

Fig. 13.2 String methods. (Part 1 of 3.)

String Method	Description
find(*substring*[**,** *start*[**,** *end*]] **)**	Returns the lowest index at which *substring* occurs in the string; returns –1 if the string does not contain *substring*. If argument *start* is specified, searching begins at that index. If argument *end* is specified, the method searches through the slice *start*:*end*.
index(*substring*[**,** *start*[**,** *end*]] **)**	Performs the same operation as **find**, but raises a **ValueError** exception if the string does not contain *substring*.
isalnum()	Returns 1 if the string contains only alphanumeric characters (i.e., numbers and letters); otherwise, returns 0.
isalpha()	Returns 1 if the string contains only alphabetic characters (i.e., letters); returns 0 otherwise.
isdigit()	Returns 1 if the string contains only numerical characters (e.g., **"0"**, **"1"**, **"2"**); otherwise, returns 0.
islower()	Returns 1 if all alphabetic characters in the string are lower-case characters (e.g., **"a"**, **"b"**, **"c"**); otherwise, returns 0.
isspace()	Returns 1 if the string contains only whitespace characters; otherwise, returns 0.
istitle()	Returns 1 if the first character of each word in the string is the only uppercase character in the word; otherwise, returns 0.
isupper()	Returns 1 if all alphabetic characters in the string are uppercase characters (e.g., **"A"**, **"B"**, **"C"**); otherwise, returns 0.
join(*sequence* **)**	Returns a string that concatenates the strings in *sequence* using the original string as the separator between concatenated strings.
ljust(*width* **)**	Returns a new string left-aligned in a whitespace string of *width* characters.
lower()	Returns a new string in which all characters in the original string are lowercase.
lstrip()	Returns a new string in which all leading whitespace is removed.
replace(*old***,** *new*[**,** *maximum*] **)**	Returns a new string in which all occurrences of *old* in the original string are replaced with *new*. Optional argument *maximum* indicates the maximum number of replacements to perform.

Fig. 13.2 String methods. (Part 2 of 3.)

String Method	Description
rfind(*substring*[**,** *start*[**,** *end*]] **)**	Returns the highest index value in which *substring* occurs in the string or –1 if the string does not contain *substring*. If argument *start* is specified, searching begins at that index. If argument *end* is specified, the method searches the slice *start*:*end*.
rindex(*substring*[**,** *start*[**,** *end*]] **)**	Performs the same operation as **rfind**, but raises a **ValueError** exception if the string does not contain *substring*.
rjust(*width* **)**	Returns a new string right-aligned in a string of *width* characters.
rstrip()	Returns a new string in which all trailing whitespace is removed.
split([*separator*] **)**	Returns a list of substrings created by splitting the original string at each *separator*. If optional argument *separator* is omitted or **None**, the string is separated by any sequence of whitespace, effectively returning a list of words.
splitlines([*keepbreaks*] **)**	Returns a list of substrings created by splitting the original string at each newline character. If optional argument *keepbreaks* is 1, the substrings in the returned list retain the newline character.
startswith(*substring*[**,** *start*[**,** *end*]] **)**	Returns 1 if the string starts with *substring*; otherwise, returns 0. If argument *start* is specified, searching begins at that index. If argument *end* is specified, the method searches through the slice *start*:*end*.
strip()	Returns a new string in which all leading and trailing whitespace is removed.
swapcase()	Returns a new string in which uppercase characters are converted to lowercase characters and lower-case characters are converted to uppercase characters.
title()	Returns a new string in which the first character of each word in the string is the only uppercase character in the word.
translate(*table*[**,** *delete*] **)**	Translates the original string to a new string. The translation is performed by first deleting any characters in optional argument *delete*, then by replacing each character *c* in the original string with the value *table***[ord(** *c* **)]**.
upper()	Returns a new string where all characters in the original string are uppercase.

Fig. 13.2 String methods. (Part 3 of 3.)

13.3 String Presentation

Strings require formatting for various reasons. For example, manipulating string presentations enables users to read and understand program instructions or output more easily. This section presents two simple examples that demonstrate string-formatting methods. Figure 13.3 uses three string methods—**`center`**, **`ljust`** and **`rjust`**—to align strings. These methods use white space characters to manipulate the string formatting.

String method **`center`** (line 6) takes one argument—an integer value—that corresponds to the total length of the output string. The method then creates a new string of this length and centers the original calling string (**`string1`**) in **`50`** spaces so that an equal number of spaces appears to the right and left of the calling string. String method **`rjust`** also aligns the **`string1`** by preceding the calling string with **`50 - len( string1 )`** space characters to right-align the string (line 7). Line 8 uses method **`ljust`** to creates a new string that is left aligned by following the calling string with **`50 - len( string1 )`** space characters. If the string is longer than the argument supplied to any of these methods, the method simply returns the original string.

Fig. 13.4 demonstrates methods that *strip* (remove) whitespace from strings. Line 4 creates a string, **`string1`**, that contains leading and trailing whitespace. String method **`strip`** removes leading and trailing whitespace from the original string (line 7). String method **`lstrip`** removes only leading whitespace (line 8) and method **`rstrip`** removes only trailing whitespace (line 9). As the output demonstrates, these methods remove all whitespace, including spaces, newlines and tabs.

```
1  # Fig. 13.3: fig13_03.py
2  # Simple output formatting example.
3
4  string1 = "Now I am here."
5
6  print string1.center( 50 )
7  print string1.rjust( 50 )
8  print string1.ljust( 50 )
```

```
                  Now I am here.
                                    Now I am here.
Now I am here.
```

Fig. 13.3 String justification.

```
1  # Fig. 13.4: fig13_04.py
2  # Stripping whitespace from a string.
3
4  string1 = "\t  \n  This is a test string. \t\t \n"
5
6  print 'Original string: "%s"\n' % string1
7  print 'Using strip: "%s"\n' % string1.strip()
8  print 'Using left strip: "%s"\n' % string1.lstrip()
9  print "Using right strip: \"%s\"\n" % string1.rstrip()
```

Fig. 13.4 Stripping whitespace from strings. (Part 1 of 2.)

```
Original string: "
  This is a test string.
"

Using strip: "This is a test string."

Using left strip: "This is a test string.
"

Using right strip: "
  This is a test string."
```

Fig. 13.4 Stripping whitespace from strings. (Part 2 of 2.)

13.4 Searching Strings

In many applications, it is necessary to search for a character or set of characters in a string. For example, a programmer creating a word processor would want to provide capabilities for searching through documents. To perform such tasks, Python provides methods such as **`find`** and **`index`**. When searching for a substring, we either can determine whether a string contains the substring, or we can retrieve the index at which a substring begins. Figure 13.5 searches for substrings at the beginning, middle and end of a string.

```
# Fig. 13.5: fig13_05.py
# Searching strings for a substring.

# counting the occurrences of a substring
string1 = "Test1, test2, test3, test4, Test5, test6"

print '"test" occurs %d times in \n\t%s' % \
   ( string1.count( "test" ), string1 )
print '"test" occurs %d times after 18th character in \n\t%s' % \
   ( string1.count( "test", 18, len( string1 ) ), string1 )
print

# finding a substring in a string
string2 = "Odd or even"

print '"%s" contains "or" starting at index %d' % \
   ( string2, string2.find( "or" ) )

# find index of "even"
try:
   print '"even" index is', string2.index( "even" )
except ValueError:
   print '"even" does not occur in "%s"' % string2

if string2.startswith( "Odd" ):
   print '"%s" starts with "Odd"' % string2

```

Fig. 13.5 Strings searched for substrings. (Part 1 of 2.)

```
28  if string2.endswith( "even" ):
29     print '"%s" ends with "even"\n' % string2
30
31  # searching from end of string
32  print 'Index from end of "test" in "%s" is %d' \
33     % ( string1, string1.rfind( "test" ) )
34  print
35
36  # find rindex of "Test"
37  try:
38     print 'First occurrence of "Test" from end at index', \
39        string1.rindex( "Test" )
40  except ValueError:
41     print '"Test" does not occur in "%s"' % string1
42
43  print
44
45  # replacing a substring
46  string3 = "One, one, one, one, one, one"
47
48  print "Original:", string3
49  print 'Replaced "one" with "two":', \
50     string3.replace( "one", "two" )
51  print "Replaced 3 maximum:", string3.replace( "one", "two", 3 )
```

```
"test" occurs 4 times in
        Test1, test2, test3, test4, Test5, test6
"test" occurs 2 times after 18th character in
        Test1, test2, test3, test4, Test5, test6

"Odd or even" contains "or" starting at index 4
"even" index is 7
"Odd or even" starts with "Odd"
"Odd or even" ends with "even"

Index from end of "test" in "Test1, test2, test3, test4, Test5, test6"
is 35

First occurrence of "Test" from end at index 28

Original: One, one, one, one, one, one
Replaced "one" with "two": One, two, two, two, two, two
Replaced 3 maximum: One, two, two, two, one, one
```

Fig. 13.5 Strings searched for substrings. (Part 2 of 2.)

Lines 5–11 use string method **`count`** to return the number of occurrences of a substring in a string or a string slice. If the method does not find the specified substring, the method returns 0. Line 8 prints the number of times the substring **`"test"`** occurs in **`string1`**. Method **`count`** takes two optional arguments that specify a slice of the string to search. Line 10 passes arguments to **`count`** that cause the method to search **`string1`** starting at index 18 (i.e., character **`"3"`**) and terminating at the end of the string. This call produces the same result as the statement

```
string1[ 18:len( string1 ) ].count( "test" )
```

but the method call with optional arguments has the added benefit of better readability and better performance, because the program does not create a new slice.

Lines 14–29 demonstrate string that search for substrings. Line 17 uses method **`find`** to return the lowest index at which the substring occurs. If a string does not contain the substring, the method returns –1. Method **`index`** (line 21) resembles method **`find`**, except that if a string does not contain the substring, the method raises a **`ValueError`** exception. A program can catch this exception and handle it appropriately, in the case that the string does not contain the specified substring.

Lines 25–29 use methods that determine whether a string begins or ends with a specific substring. If the string begins with the substring, method **`startswith`** returns 1 (line 25). This call produces the same result as the expression

```
string2[ 0:len( "Odd" ) ] == "Odd"
```

If a string ends with the substring, method **`endswith`** returns 1 (line 28). Using this method produces the same result as the expression

```
string2[ -len( "even" ): ] == "even"
```

The program can search for a substring starting from the end of a string. Lines 32–43 use methods **`rfind`** and **`rindex`** to determine whether **`string1`** contains certain substrings. Method **`rfind`** returns the index of the first occurrence of the substring searching from the end of the string. If the method does not find the substring, it returns –1. Method **`rindex`** returns the highest index at which the substring begins and raises a **`ValueError`** if the method does not find the substring. Our program catches the exception to handle the case where the string does not contain the specified substring.

At times, a user may want to find substring to perform an action on that substring. For example, a user may perform a search for a current phrase in a document and replace that phrase with another phrase. Method **`replace`** takes two substrings and searches a document for the first substring then replaces that substring with the substring in the second argument. Line 50 replaces all occurrences of the substring **`"one"`** in **`string3`** with the substring **`"two"`**. Method **`replace`** takes an optional third argument that sets the maximum number of replacements. Line 51 replaces up to three occurrences of substring **`"one"`** with substring **`"two"`**.

13.5 Joining and Splitting Strings

A computer processes code in much the same way people process text when reading. When you read a sentence, your brain breaks the sentence into individual words, or *tokens*, each of which conveys a meaning. This process is known as tokenization. Interpreters perform tokenization because they break up statements into such individual components as keywords, identifiers, operators and other elements of a programming language. Tokens are separated by delimiters, typically whitespace characters such as blank, tab, newline and carriage return. Other characters also may be used as delimiters to separate tokens. In this section, we study string methods that perform delimiter-based string splitting and joining.

Figure 13.6 demonstrates string methods **`split`** and **`join`**. Line 5 creates **`string1`**, a comma-separated string of letters. Lines 7–11 demonstrate how to split a string into tokens using delimiters. Line 8 calls method **`split`** with no arguments, which splits the string at each occurrence of a whitespace character. The method returns a list of tokens and

```
1  # Fig. 13.6: fig13_06.py
2  # Token splitting and delimiter joining.
3
4  # splitting strings
5  string1 = "A, B, C, D, E, F"
6
7  print "String is:", string1
8  print "Split string by spaces:", string1.split()
9  print "Split string by commas:", string1.split( "," )
10 print "Split string by commas, max 2:", string1.split( ",", 2 )
11 print
12
13 # joining strings
14 list1 =  [ "A", "B", "C", "D", "E", "F" ]
15 string2 = "___"
16
17 print "List is:", list1
18 print 'Joining with "%s": %s' \
19    % ( string2, string2.join ( list1 ) )
20 print 'Joining with "-.-":', "-.-".join( list1 )
```

```
String is: A, B, C, D, E, F
Split string by spaces: ['A,', 'B,', 'C,', 'D,', 'E,', 'F']
Split string by commas: ['A', ' B', ' C', ' D', ' E', ' F']
Split string by commas, max 2: ['A', ' B', ' C, D, E, F']

List is: ['A', 'B', 'C', 'D', 'E', 'F']
Joining with "___": A___B___C___D___E___F
Joining with "-.-": A-.-B-.-C-.-D-.-E-.-F
```

Fig. 13.6 Splitting and joining strings.

the program displays the tokens. In line 9, method **`split`** receives the argument **`","`**, which represents the delimiter (the string is split at each occurrence of a comma). In line 10, method **`split`** receives two arguments—the delimiter and an integer value that specifies the maximum number of splits to perform.

Given a list of tokens, method **`join`** combines the list with a pre-defined delimiter. Line 14 creates a list of letter tokens and line 15 creates a delimiter **`string2`** that contains three underscore (**`"_"`**) characters. Lines 18–19 show the results of calling **`string2`**'s **`join`** method. The method receives the list of tokens as an argument and returns a string where the tokens are joined by the underscore delimiter in **`string2`**. Line 20 demonstrates combining the **`print`** method with a call to a string's **`join`** method.

Performance Tip 13.1

*When building a complex string, it is more efficient to include the pieces in a list and then use method **`join`** to assemble the string, rather than using the concatenation (**`+`**) operator.*

13.6 Regular Expressions

String methods allow programs to search for a specific substring. For instance, to determine whether a string contains the substrings representing the days of the week (**`"Monday"`**,

"Tuesday", **"Wednesday"**, etc.), the program can invoke string method **find** for each substring (i.e., the program needs to invoke method **find** seven times to search for every day of the week). Depending on the search, a program may need to invoke method **find** numerous time, an inefficient way to solve a problem. *Regular expressions* provide a more efficient and powerful alternative. A regular expression is a *text pattern* that a program uses to find substrings that match patterns. In the remainder of this chapter, we discuss Python's various regular-expression capabilities.

Good Programming Practice 13.1

Use string methods where only simple processing is required. This prevents errors caused by the more complex regular expressions and increases program readability.

We begin our discussion with a simple example (Fig. 13.7) in which we search various welcoming phrases for **"hello"**, **"Hello"** and **"world!"**.

Line 4 imports the regular-expression module ***re***, which provides regular-expression processing capabilities in Python. List **testStrings** (line 7) contains the strings that are searched with the regular expressions created in line 8. Note that the regular expressions closely resemble the strings.

```
1  # Fig. 13.7: fig13_07.py
2  # Simple regular-expression example.
3
4  import re
5
6  # list of strings to search and expressions used to search
7  testStrings = [ "Hello World", "Hello world!", "hello world" ]
8  expressions = [ "hello", "Hello", "world!" ]
9
10 # search every expression in every string
11 for string in testStrings:
12
13    for expression in expressions:
14
15       if re.search( expression, string ):
16          print expression, "found in string", string
17       else:
18          print expression, "not found in string", string
19
20    print
```

```
hello not found in string Hello World
Hello found in string Hello World
world! not found in string Hello World

hello not found in string Hello world!
Hello found in string Hello world!
world! found in string Hello world!

hello found in string hello world
Hello not found in string hello world
world! not found in string hello world
```

Fig. 13.7 Regular-expression example.

The remainder of the program consists of a nested **for** loop that tests each regular expression in list **expressions** against each string in list **testStrings**. Function ***re.search*** looks for the first occurrence of a regular expression in a string and returns an object that contains the substring matching the regular expression. If the string does not contain the pattern, **re.search** returns **None**. The program determines whether the function call returns a value, then prints an appropriate message. We discuss how to use the object returned by **re.search** in the next section.

Each regular expression in this example is a substring of one of the test strings. In fact, line 15 could be replaced with the expression

```
if string.find( expression ) >= 0:
```

and the program would produce the same result. In the remaining sections, we explore how to create more powerful regular-expression pattern strings.

13.7 Compiling Regular Expressions and Manipulating Regular Expression Objects

This section examines *compiled regular-expression objects* and introduces the object returned by function **re.search**, which contains the results of a search. The **re** module normally compiles a regular expression into a form that the module uses to process a string. If a program uses the same regular expression several times, compiling the regular expression in advance may make the program more efficient. Figure 13.8 demonstrates how to compile regular expressions in advance to create compiled regular-expression objects and shows how to use the object returned from **re.search** to view the search results.

Function ***re.compile*** (line 11) takes as an argument a regular expression and returns an ***SRE_Pattern*** *object* that represents a compiled regular expression. Compiled regular expression objects provide all the functionality available in module **re**. For example, compiled-expression object method **search** (line 22) corresponds to function **re.search** (line 20). As the output demonstrates, both approaches return an ***SRE_Match*** *object*. This object supports various methods for retrieving the results of regular-expression processing. Method ***group*** (lines 26–28) returns the substring that matches the pattern. We discuss this method further when we discuss grouping in Section 13.11.

```
1   # Fig. 13.08: fig13_08.py
2   # Compiled regular-expression and match objects.
3
4   import re
5
6   testString = "Hello world"
7   formatString = "%-35s: %s"   # string for formatting the output
8
9   # create regular expression and compiled expression
10  expression = "Hello"
11  compiledExpression = re.compile( expression )
12
13  # print expression and compiled expression
14  print formatString % ( "The expression", expression )
```

Fig. 13.8 Regular-expression compilation. (Part 1 of 2.)

```
print formatString % ( "The compiled expression",
   compiledExpression )

# search using re.search and compiled expression's search method
print formatString % ( "Non-compiled search",
   re.search( expression, testString ) )
print formatString % ( "Compiled search",
   compiledExpression.search( testString ) )

# print results of searching
print formatString % ( "search SRE_Match contains",
   re.search( expression, testString ).group() )
print formatString % ( "compiled search SRE_Match contains",
   compiledExpression.search( testString ).group() )
```

```
The expression                      : Hello
The compiled expression            : <SRE_Pattern object at 0x00B60A20>
Non-compiled search                 : <SRE_Match object at 0x00D0F9B8>
Compiled search                     : <SRE_Match object at 0x00D0F9B8>
search SRE_Match contains           : Hello
compiled search SRE_Match contains : Hello
```

Fig. 13.8 Regular-expression compilation. (Part 2 of 2.)

13.8 Regular Expression Repetition and Placement Characters

We now begin our discussion on how to build more sophisticated pattern strings. Regular expressions are like a "language within a language." Much as Python has a strictly defined syntax for creating programs, regular expressions specify several characters for creating patterns. Most patterns are built using a combination of characters, *metacharacters* and *escape sequences*. A metacharacter is a regular-expression syntax element, just as keyword **if** is a Python syntax element. Characters match themselves. A metacharacter's task is to repeat, group, place or classify one or more characters. This section introduces metacharacters for repeating. We discuss escape sequences and other metacharacters in the following sections.

Figure 13.9 demonstrates the basic repetition metacharacters—**?**, **+** and *****. Line 7 creates a list of regular expressions that contain these symbols. Metacharacter **?** matches exactly zero or one occurrences of the *expression* it follows. An expression can be a single character, an escape sequence, a class of characters (discussed in Section 13.9) or a group (discussed in Section 13.11). In our simple example, we use only a single character for all the expressions that precede a repeating metacharacter. For example, the first regular expression in line 7, **"Hel?o"** matches the letter **H**, followed by the letter **e**, followed by zero or one occurrences of the letter **l**, followed by the letter **o**.

```
1  # Fig. 13.9: fig13_09.py
2  # Repetition patterns, matching vs searching.
3
4  import re
5
```

Fig. 13.9 Searching and matching strings with repetition metacharacters. (Part 1 of 2.)

```python
testStrings = [ "Heo", "Helo", "Hellllo" ]
expressions = [ "Hel?o", "Hel+o", "Hel*o" ]

# match every expression with every string
for expression in expressions:

   for string in testStrings:

      if re.match( expression, string ):
         print expression, "matches", string
      else:
         print expression, "does not match", string

   print

# demonstrate the difference between matching and searching
expression1 = "elo"    # plain string
expression2 = "^elo"   # "elo" at beginning of string
expression3 = "elo$"   # "elo" at end of string

# match expression1 with testStrings[ 1 ]
if re.match( expression1, testStrings[ 1 ] ):
   print expression1, "matches", testStrings[ 1 ]

# search for expression1 in testStrings[ 1 ]
if re.search( expression1, testStrings[ 1 ] ):
   print expression1, "found in", testStrings[ 1 ]

# search for expression2 in testStrings[ 1 ]
if re.search( expression2, testStrings[ 1 ] ):
   print expression2, "found in", testStrings[ 1 ]

# search for expression3 in testStrings[ 1 ]
if re.search( expression3, testStrings[ 1 ] ):
   print expression3, "found in", testStrings[ 1 ]
```

```
Hel?o matches Heo
Hel?o matches Helo
Hel?o does not match Hellllo

Hel+o does not match Heo
Hel+o matches Helo
Hel+o matches Hellllo

Hel*o matches Heo
Hel*o matches Helo
Hel*o matches Hellllo


elo found in Helo
elo$ found in Helo
```

Fig. 13.9 Searching and matching strings with repetition metacharacters. (Part 2 of 2.)

Metacharacter **+** matches one or more occurrences of the expression it follows. For example, the second regular expression in line 7, **"Hel+o"**, matches the letter **H**, followed by the letter **e**, followed by one or more occurrences of the letter **l**, followed by the letter **o**. Metacharacter ***** matches zero or more occurrences of the expression it follows. For example, the third regular expression in line 7, **"Hel*o"**, matches the letter **H**, followed by the letter **e**, followed by zero or more occurrences of the letter **l**, followed by the letter **o**.

Lines 10–19 contain a nested **for** loop that applies each regular expression from line 7 to each string from line 6. Function ***re.match*** (line 14) matches an expression to a string. Unlike function **re.search** (which returns an **SRE_Match** object if any part of the string matches the expression), function **re.match** returns an **SRE_Match** object only if the beginning of the string matches the regular expression.

Regular expressions can contain two additional metacharacters to place a pattern within a string. Metacharacter **^** indicates placement at the beginning of the string; metacharacter **$** indicates placement at the end of the string. A search or a match returns a value only if a string contains the specified pattern at the beginning or end of the string, respectively. Lines 22–40 create regular expressions that contain these metacharacters and use functions **re.match** and **re.search** to process a string.

The regular expressions in lines 22–24 correspond to the sequence of characters **"elo"** anywhere in a string, at the beginning of a string and at the end of a string, respectively. Notice, from the output that function **re.match** returns **None** when passed arguments **expression1** and **testStrings[2]**, because the regular expression **"elo"** does not match the entire string **"Helo"**. Similarly, function **re.search** returns **None** when passed arguments **expression2** and **testStrings[1]**, because the string **"elo"** does not appear at the beginning of the string **"Helo"**.

13.9 Classes and Special Sequences

In this section, we explore two more basic regular-expression building blocks—*character classes* and *special sequences*. A character class specifies a group of characters to match in a string. A special sequence is a shortcut for a common class of characters.

The metacharacters **[** and **]** denote a *regular expression class*. A regular expression that contains a class matches one character in the class. For example, the regular expression class **"[abc]"** matches the letter **a**, the letter **b** or the letter **c**. Classes can use the **-** character to specify a range of consecutive characters. For example, the regular expression **"[a-d]"** is identical to the regular expression **"[abcd]"**.

When placed at the beginning of a class, the metacharacter **^** *negates* the class. This means that the regular expression matches all characters *except* those specified in the class. For example, the class **"[^a-c]"** matches any character but **a**, **b** and **c**. Special sequences (Fig. 13.10) describe commonly used classes of characters.

Figure 13.11 demonstrates regular expressions that contain classes and special sequences. We also demonstrate how to avoid common regular-expression errors. The regular expression in line 8 contains one new metacharacter and demonstrates an important point about using regular expressions. The metacharacter **|** matches either the regular expression to the left or to the right of the metacharacter. Another way to write the expression **"[abc]"** is **"a|b|c"**. Thus, the regular expression in line 8 matches either the string **"2x+5y"** or the string **"7y-3z"**.

Special Sequence	Describes
`\d`	The class of digits (`[0-9]`).
`\D`	The negation of the class of digits (`[^0-9]`).
`\s`	The whitespace characters class (`[ \n\f\r\t\v]`).
`\S`	The negation of the whitespace characters class (`[^ \n\f\r\t\v]`).
`\w`	The alphanumeric characters class (`[a-zA-Z0-9_]`).
`\W`	The negation of the alphanumeric characters class (`[^a-zA-Z0-9_]`).
`\\`	The backslash (`\`).

Fig. 13.10 Regular-expression special sequences.

```
# Fig. 13.11: fig13_11.py
# Program that demonstrates classes and special sequences.

import re

# specifying character classes with [ ]
testStrings = [ "2x+5y","7y-3z" ]
expressions = [ r"2x\+5y|7y-3z",
                r"[0-9][a-zA-Z0-9_].[0-9][yz]",
                r"\d\w-\d\w" ]

# match every expression with every string
for expression in expressions:

   for testString in testStrings:

      if re.match( expression, testString ):
         print expression, "matches", testString

# specifying character classes with special sequences
testString1 = "800-123-4567"
testString2 = "617-123-4567"
testString3 = "email: \t joe_doe@deitel.com"

expression1 = r"^\d{3}-\d{3}-\d{4}$"
expression2 = r"\w+:\s+\w+@\w+\.(com|org|net)"

# matching with character classes
if re.match( expression1, testString1 ):
   print expression1, "matches", testString1

if re.match( expression1, testString2 ):
   print expression1, "matches", testString2

if re.match( expression2, testString3 ):
   print expression2, "matches", testString3
```

Fig. 13.11 Regular expressions with classes and special sequences. (Part 1 of 2.)

```
2x\+5y|7y-3z matches 2x+5y
2x\+5y|7y-3z matches 7y-3z
[0-9][a-zA-Z0-9_].[0-9][yz] matches 2x+5y
[0-9][a-zA-Z0-9_].[0-9][yz] matches 7y-3z
\d\w-\d\w matches 7y-3z
^\d{3}-\d{3}-\d{4}$ matches 800-123-4567
^\d{3}-\d{3}-\d{4}$ matches 617-123-4567
\w+:\s+\w+@\w+\.(com|org|net) matches email:     joe_doe@deitel.com
```

Fig. 13.11 Regular expressions with classes and special sequences. (Part 2 of 2.)

Notice that **** the escape metacharacter, precedes the character **+** in the regular expression at line 8. This matches the character **+**, rather than using the repetition metacharacter **+**. If the **+** was not escaped, the regular expression would match one or more **x** characters, followed by the numeric character **5** (as shown in Fig. 13.12).

Note also that the regular expression in line 8 is a *raw string*—i.e., a string created by preceding the string with the character **r**. Usually, when a **** appears in a string, Python interprets this character as an escape character and attempts to replace the **** and the character that follows with the correct escape sequence. When a **** appears within a raw string, Python does not interpret the character as the escape character, but instead interprets the character as the literal backslash character. For example, Python interprets the string **"\n"** as one newline character, but it interprets the string **r"\n"** as two characters—a backslash and the character **n**.

Common Programming Error 13.1

Placing a backslash at the end of a raw string results in a syntax error.

Good Programming Practice 13.2

Regular expression pattern strings often contain backslash characters. Using raw strings to create pattern strings eliminates the need to escape each backslash in the pattern string, thus making the pattern string easier to read.

Lines 9–10 create two additional regular expressions. The metacharacter **.** matches any character in a string except for a newline. The regular expression in line 9 matches a digit, followed by an alphanumeric character, followed by any character except a newline,

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license" for more information.
>>> import re
>>> print re.match( "2x+5y", "2x+5y" )
None
>>> print re.match( "2x+5y", "2x5y" )
<SRE_Match object at 0x00932268>
>>> print re.match( "2x+5y", "2xx5y" )
<SRE_Match object at 0x00949A88>
```

Fig. 13.12 \ metacharacter in regular expressions.

followed by a digit, followed by the letter **y** or the letter **z**. The regular expression in line 10 uses special sequences to create a similar regular expression. This expression matches a digit, followed by an alphanumeric character, followed by the character **-**, followed by a digit, followed by an alphanumeric character. Lines 13–18 contain a nested **for** loop that attempts to match each expression from lines 8–10 to each string in line 7.

The remainder of the program creates more complex regular expressions. The metacharacters **{** and **}** provide another way to repeat characters. The expression in line 25 matches three digits (as specified between curly brackets), followed by the character **-**, three digits, another **-** and four digits. By placing the regular expression between metacharacters **^** and **$**, we specify that we want the regular expression to match the entire string. We can also use the bracket metacharacters to specify a range of repetitions. For example, the expression **"\d{1,3}"** matches one, two or three digits.

Line 26 creates a regular expression that matches one or more alphanumeric characters, followed by a colon (**:**), followed by one or more whitespace characters, followed by one or more alphanumeric characters, followed by the **@** character, followed by one or more alphanumeric characters, followed by the **.** character (notice the backslash to escape this regular expression character), followed by the sequence of characters **com**, **org** or **net**. The remainder of the program attempts to match the regular expressions to the test strings.

13.10 Regular Expression String-Manipulation Functions

The first few sections of this chapter discussed basic string methods for string manipulation and promised a more powerful version of these methods that use regular expressions. Module **re** provides pattern-based, string-manipulation capabilities, such as substituting a substring in a string and splitting a string with a delimiter. Figure 13.13 demonstrates these capabilities.

```
# Fig. 13.13: fig13_13.py
# Regular-expression string manipulation.

import re

testString1 = "This sentence ends in 5 stars *****"
testString2 = "1,2,3,4,5,6,7"
testString3 = "1+2x*3-y"
formatString = "%-34s: %s"   # string to format output

print formatString % ( "Original string", testString1 )

# regular expression substitution
testString1 = re.sub( r"\*", r"^", testString1 )
print formatString % ( "^ substituted for *", testString1 )

testString1 = re.sub( r"stars", "carets", testString1 )
print formatString % ( '"carets" substituted for "stars"',
   testString1 )

print formatString % ( 'Every word replaced by "word"',
   re.sub( r"\w+", "word", testString1 ) )
```

Fig. 13.13 Regular-expression-based string manipulation. (Part 1 of 2.)

```
23
24  print formatString % ( 'Replace first 3 digits by "digit"',
25     re.sub( r"\d", "digit", testString2, 3 ) )
26
27  # regular expression splitting
28  print formatString % ( "Splitting " + testString2,
29     re.split( r",", testString2 ) )
30
31  print formatString % ( "Splitting " + testString3,
32     re.split( r"[+\-*/%]", testString3 ) )
```

```
Original string                  : This sentence ends in 5 stars *****
^ substituted for *              : This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars" : This sentence ends in 5 carets ^^^^^
Every word replaced by "word"    : word word word word word word ^^^^^
Replace first 3 digits by "digit" : digit,digit,digit,4,5,6,7
Splitting 1,2,3,4,5,6,7          : ['1', '2', '3', '4', '5', '6', '7']
Splitting 1+2x*3-y                : ['1', '2x', '3', 'y']
```

Fig. 13.13 Regular-expression-based string manipulation. (Part 2 of 2.)

Function ***re.sub*** (line 14) takes three arguments. The second argument is a substring that is substituted for every substring in the third argument that matches the pattern described by the first argument. Line 14 substitutes the caret character (**^**) for the asterisk character (*****) in string **testString1**. To replace the asterisk character, the method must use the regular expression **"*"**, because ***** is a metacharacter. Lines 21–22 replace every word (**"\w+"**) by the substring **"word"**. Lines 24–25 use the function's optional fourth argument to specify a maximum number of replacements to perform.

Function ***re.split*** takes two arguments. The first argument is a regular expression that describes a pattern delimiter. The function returns a list of tokens created by splitting the second argument at the delimiter. Lines 28–29 print the results of splitting variable **testString2** on commas (**,**). Line 32 calls **re.split**, passing a delimiter pattern that matches one of five mathematical operators. Notice that this regular expression defines a class and escapes the **-** character, but not the ***** character. This demonstrates a subtle regular expression feature. When any character—except **^** (for negation) or **-** (for a range)—appears inside a class that character is interpreted literally as the character. Therefore, metacharacters such as **$**, **+** or ***** do not need to be escaped when they appear inside a class.

13.11 Grouping

In Fig. 13.8, we saw how a program can use method **group** to extract matching substrings from an **SRE_Match** object. This method arises from a more sophisticated regular-expression technique—*grouping*. A regular expression may specify groups of substrings to match in a string. A program then searches or matches a string with the regular expression and extracts information from the matching groups. Figure 13.14 creates regular expressions with groups and prints the information extracted from these groups.

The regular expression in line 12 describes three groups. The metacharacters **(** and **)** denote a group. The first group matches a word (**\w+**), followed by a space, followed by another word. The second group matches three digits, followed by the character **-**, followed

```
1  # Fig. 13.14: fig13_14.py
2  # Program that demonstrates grouping and greedy operations.
3
4  import re
5
6  formatString1 = "%-22s: %s"   # string to format output
7
8  # string that contains fields and expression to extract fields
9  testString1 = \
10    "Albert Antstein, phone: 123-4567, e-mail: albert@bug2bug.com"
11 expression1 = \
12    r"(\w+ \w+), phone: (\d{3}-\d{4}), e-mail: (\w+@\w+\.\w{3})"
13
14 print formatString1 % ( "Extract all user data",
15    re.match( expression1, testString1 ).groups() )
16 print formatString1 % ( "Extract user e-mail",
17    re.match( expression1, testString1 ).group( 3 ) )
18 print
19
20 # greedy operations and grouping
21 formatString2 = "%-38s: %s"   # string to format output
22
23 # strings and patterns to find base directory in a path
24 pathString = "/books/2001/python"  # file path string
25
26 expression2 = "(/.+)/"  # greedy operator expression
27 print formatString1 % ( "Greedy error",
28    re.match( expression2, pathString ).group( 1 ) )
29
30 expression3 = "(/.+?)/"  # non-greedy operator expression
31 print formatString1 % ( "No error, base only",
32    re.match( expression3, pathString ).group( 1 ) )
```

```
Extract all user data : ('Albert Antstein', '123-4567', 'albert@
bug2bug.com')
Extract user e-mail   : albert@bug2bug.com

Greedy error          : /books/2001
No error, base only   : /books
```

Fig. 13.14 Regular-expression groups and greedy operators.

by four digits. The third group matches one or more alphanumeric characters, followed by the character **@**, followed by one or more alphanumeric characters, followed by the character **.**, followed by three alphanumeric characters. This regular expression matches the string in variable **testString1**. The three groups match the name, phone number and e-mail address of the person, respectively.

Lines 14–17 demonstrate the benefits of grouping. Line 15 calls function **re.match**, which returns an **SRE_Match** object. This object's ***groups*** method returns a list of substrings. Each substring in the list corresponds to the substring that matches a group in the regular expression. The first substring in the list matches the first group in the regular expression, and so on. The result of line 15 is that the program obtains the person's name, phone number

and e-mail address. Line 17 calls the **SRE_Match**'s method **group**, passing integer value 3 as an argument. This call returns the substring that matches the third group in the regular expression, which retrieves the e-mail address substring from **testString1**.

Regular-expression grouping introduces another subtle regular-expression issue. The metacharacters **+** and ***** are called *greedy operators*. A greedy operator attempts to match as many characters as possible. Sometimes this is not the desired behavior. Lines 20–32 demonstrate the problem of greedy operators. Line 24 is a string that contains a sample path that might be part of a URL. Suppose we wish to write a regular expression that obtains the root directory name from the path (i.e., **/books** in this example). Lines 26–28 attempt this operation, but fail because of the greedy behavior of the **+** operator in **expression2**. When an operator is greedy the regular expression module tries to match as much of the expression that precedes the operator as possible. Initially, this causes **expression2** to match the entire string. However, the regular expression module must allow for the rest of the pattern to be matched. In this case, the group that contains the greedy operator must be immediately followed by a slash (**/**) as specified in **expression2**. Therefore, the regular expression module searches backwards in the string until the regular expression module can guarantee that there is a slash (**/**) that will follow the initial group in **expression2**. Thus, the group that contains the greedy operator matches **/books/2001**.

The regular expression in line 30 modifies the behavior of the greedy **+** operator to obtain the root directory name in the sample path correctly. Placing the **?** metacharacter after the greedy **+** operator changes the behavior of the **+** operator. Now, when the regular expression module searches the string using **expression3**, the module searches one character at a time until it finds the smallest string that matches the pattern in the group (i.e., **/books**).

This chapter presented basic string manipulation capabilities, as well as how to use the powerful regular-expression-processing capabilities of module **re**. Chapter 14 introduces file processing, which enables programs to read information from files on disk and write information to files on disk. Many programs that process files use regular expressions and other string-processing capabilities to search and manipulate the contents of those files.

13.12 Internet and World Wide Web Resources

The following resources offer more information about the complex topic of regular expressions.

etext.lib.virginia.edu/helpsheets/regex.html
This tutorial discusses common regular expression uses, writing complex regular expressions and other topics like escape characters and anchors.

www.zvon.org/other/reReference/Output
This reference describes common regular expression special sequences.

py-howto.sourceforge.net/regex/regex.html
This tutorial discusses using regular expressions and the **re** module. Topics covered include common problems, modifying strings and pattern matching.

www.devshed.com/Server_Side/Administration/RegExp
This article describes common uses of regular expressions.

py-howto.sourceforge.net/regex/regex.html
This document contains information about regular-expression processing with module **re**, and discusses greedy operators and how to use raw strings as regular-expression pattern strings.

SUMMARY

- Python represents strings as sequences of characters. Characters are the fundamental building blocks of Python source programs. Every program is composed of a sequence of characters that—when they are grouped together meaningfully—is interpreted by the computer as a series of instructions used to accomplish a task.
- Each character has a corresponding character code (sometimes called its integer ordinal value) the ASCII or Unicode character set.
- Python supports strings as a basic data type.
- Strings are immutable sequences—once a string is created, it cannot be changed.
- Methods **`center`**, **`ljust`** and **`rjust`** control how a string is output by "padding" the string with space characters. Method **`center`** takes one argument, which corresponds to the length of the output string. The method then creates a new string of this length and centers the calling string in the specified number of spaces. Method **`rjust`** right-justifies the calling string by preceding the string with the difference of the specified number of spaces and the length of the calling string. Method **`ljust`** creates a new string where the calling string is followed by the difference of the specified number of spaces and the length of the calling string.
- String method **`strip`** removes leading and trailing whitespace from the calling string. String method **`lstrip`** removes only leading whitespace. Method **`rstrip`** removes only trailing whitespace.
- String method **`count`** returns the number of times the substring occurs in the string. If the method does not find the specified substring, the method returns 0.
- Method **`find`** returns the lowest index in the string that begins the specified substring. If the string does not contain the specified substring, the method returns –1.
- Method **`index`** is similar to method **`find`**. However, if the method does not find the specified substring, the method raises a **`ValueError`** exception.
- Method **`startswith`** returns 1 if the string begins with the specified substring.
- Method **`endswith`** returns 1 if the string ends with the specified substring.
- Method **`rfind`** is similar to method **`find`**, except the former returns the highest index at which the specified substring begins. If the method does not find the specified substring, it returns –1.
- Method **`rindex`** is similar to method **`index`**, except that **`rindex`** returns the highest index at which the specified substring begins (and raises a **`ValueError`** if the method does not find the substring).
- Method **`replace`** receives two substrings as arguments. The method searches the calling string for the first substring and replaces that substring with the second argument. Method **`replace`** takes an optional third argument that specifies the maximum number of replacements.
- When you read a sentence, your mind breaks the sentence into words, or tokens, each of which conveys meaning to you. Interpreters also perform tokenization. They break up statements into individual pieces like keywords, identifiers, operators and other elements of a programming language.
- Tokens are separated from one another by delimiters, typically whitespace characters such as blank, tab, newline and carriage return. Other characters also may be used as delimiters to separate tokens.
- Method **`split`** returns a list of tokens. When a call to method **`split`** passes no arguments, the method splits the string by any whitespace. The method takes an optional second argument that specifies the maximum number of splits to perform.
- Given a list of tokens, method **`join`** joins that list with a delimiter. The method receives the list of tokens as an argument and returns a string where the tokens are joined by the delimiter specified in the calling string.

- A regular expression is a text pattern that a program uses to find substrings that match patterns.
- The **re** regular-expression module provides regular expression capability in Python.
- Function **re.search** looks for the first occurrence of a regular expression in a string and returns an object that contains the substring matching the regular expression. If the string does not contain the pattern, **re.search** returns **None**.
- Compiling regular expressions can make programs more efficient. To use a regular expression, the **re** module first compiles the expression into a form that the module uses to process a string.
- Function **re.compile** takes as an argument a regular expression and returns an **SRE_Pattern** object that represents a compiled regular expression. Compiled regular expression objects provide all the functionality available in module **re**.
- **SRE_Match** methods enable a program to retrieve the results of regular-expression processing.
- Most patterns are built using a combination of characters, metacharacters and escape sequences. A metacharacter is a regular-expression syntax element. A metacharacter's job is to repeat, group, place or classify.
- Metacharacter **?** matches exactly zero or one occurrences of the expression it follows. Metacharacter **+** matches one or more occurrences of the expression it follows. Metacharacter ***** matches zero or more occurrences of the expression it follows.
- Function **re.match** matches an expression to a string. Unlike function **re.search** (which returns an **SRE_Match** object if any part of the string matches the expression), function **re.match** returns an **SRE_Match** object only if the beginning of the string matches the regular expression.
- Metacharacter **^** indicates placement at the beginning of the string; metacharacter **$** indicates placement at the end of the string.
- A character class specifies a group of characters to match in a string.
- A special sequence is a shortcut for a common class of characters.
- The metacharacters **[** and **]** denote a regular expression class. A regular expression that contains a class matches one character in the class.
- Classes can use the **-** character to specify a range of consecutive characters.
- When placed within a class as the first character after the square bracket, metacharacter **^** negates the class—the regular expression matches all characters except those specified in the class.
- The metacharacter **|** matches either the regular expression to the left of the **|** or the regular expression to the right.
- A raw string is created by preceding the string with the character **r**.
- Usually, when a **** appears in a string, Python interprets this character as an escape character and attempts to replace the **** and the character that follows with the correct escape sequence.
- When a **** appears within a raw string, Python does not interpret the character as the escape character, but instead as the literal backslash character.
- The metacharacter **.** matches any character in a string except for a newline.
- The metacharacters **{** and **}** provide another way to repeat characters.
- By placing a regular expression between metacharacters **^** and **$**, we specify that we want the regular expression to match the entire string.
- Module **re** provides pattern-based, string-manipulation capabilities, such as substituting a substring in a string and splitting a string with a delimiter.
- Function **re.sub** takes three arguments. The second argument is a substring that is substituted for every substring in the third argument that matches the pattern described by the first argument. The function's optional fourth argument specifies a maximum number of replacements to perform.

- Function **re.split** takes two arguments. The first argument is a regular expression that describes a pattern delimiter. The function returns a list of tokens created by splitting the second argument at the delimiter.
- If metacharacters such as **$**, **+** or ***** appear inside a class, they do not need to be escaped.
- Method **group** extracts matching substrings from an **SRE_Match** object. A regular expression may specify groups of substrings to match in a string. The metacharacters **(** and **)** denote a group.
- Function **re.match** returns an **SRE_Match** object. This object's **groups** method returns a list of substrings. Each substring in the list corresponds to the substring that matches a group in the regular expression. The first substring in the list matches the first group in the regular expression, and so on.
- The metacharacters **+** and ***** are called greedy operators. A greedy operator attempts to match as many characters as possible.

TERMINOLOGY

$ metacharacter
% metacharacter
(metacharacter
) metacharacter
***** metacharacter
+ metacharacter
. metacharacter
? metacharacter
**** metacharacter
[metacharacter
] metacharacter
^ metacharacter
{ metacharacter
} metacharacter
| metacharacter
capitalize method
character
character class
center method
count method
delimiter
encode method
endswith method
escape sequence
expandtabs method
find method
greedy operator
group method
groups method
index method
integer ordinal value
isalnum method
isalpha method
isdigit method
islower method
isspace method
istitle method
isupper method
join method
ljust method
lower method
lstrip method
metacharacter
ord function
raw string
re module
re.compile function
re.match function
re.search function
re.split function
re.sub function
regular expression
replace method
rfind method
rindex method
rjust method
rstrip method
search method
split method
splitlines method
SRE_MATCH object
SRE_Pattern object
startswith method
string
strip method
swapcase method
title method
token
tokenization
translate method
upper method
white space character

SELF REVIEW EXERCISES

13.1 Fill in the blanks in each of the following:

a) Method ________ returns a new string where all leading and trailing whitespace has been removed.
b) Python represents strings as sequences of ________.
c) Method ________ returns the number of times a specified substring occurs in a string.
d) Tokens are separated from one another by ________.
e) A ________ is a text pattern that a program uses to process strings.
f) Function ________ looks for the first occurrence of a regular expression in a string.
g) The task of a ________ is to repeat, group, place or classify one or more characters.
h) Method ________ takes a regular expression as an argument and returns an object that represents a compiled regular expression.
i) Compiled regular expressions provide all the functionality available in module ________.
j) The metacharacters ________ and ________ denote a regular expression class.

13.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) String method **`capitalize`** returns a new string where the first character of each word in the string is the one and only uppercase character in the word.
b) String method **`find`** searches a string for a substring and raises a **`ValueError`** exception if the string does not contain a substring.
c) Method **`rindex`** returns the highest index at which the specified substring begins.
d) Most string methods modify the string in-place.
e) Any string can be treated as a regular expression.
f) Metacharacter **`?`** matches exactly one occurrence of the expression it follows.
g) Method **`group`** returns an **`SRE_Match`** object.
h) Method **`re.match`** does not search through a string, but returns a match object only if the string matches the specified regular expression starting from the beginning.
i) The class **`[^0-9]`** matches any digit but **`0`**.
j) Preceding a string with the character **`r`** creates a raw string.

ANSWERS TO SELF REVIEW EXERCISES

13.1 a) **`strip`**. b) characters. c) **`count`**. d) delimiters. e) regular expression. f) **`re.search`**. g) metacharacter. h) **`re.compile`**. i) **`re`**. j) **`[`**, **`]`**.

13.2 a) False. String method **`title`** returns a new string where the first character of each word in the string is the one and only uppercase character in the word. b) False. String method **`index`** searches a string for a substring and raises a **`ValueError`** exception if the string does not contain a substring. c) True. d) False. Strings are immutable, so string methods that appear to modify a string actually return a new string. e) True. f) False. Metacharacter **`?`** matches exactly zero or one occurrences of the expression it follows. g) False. Method **`group`** returns the substring that matches a regular expression. h) True. i) False. The class **`[^0-9]`** excludes all digits. j) True.

EXERCISES

13.3 Use a regular expression to count the number of digits, non-digit characters, whitespace characters and words in a string.

13.4 Use a regular expression to search through an XHTML string and to locate all valid URLs. For the purpose of this exercise, assume that a valid URL is enclosed in quotes and begins with **`http://`**.

13.5 Write a regular expression that searches a string and matches a valid number. A number can have any number of digits, but it can have only digits and a decimal point. The decimal point is optional, but if it appears in the number, there must be only one, and it must have digits on its left and its right. There should be whitespace or a beginning or end-of-line character on either side of a valid number. Negative numbers are preceded by a minus sign.

13.6 Write a program that receives XHTML as input and outputs the number of XHTML tags in the string. The program should count the number of tags nested at each level. For example, the XHTML:

```
<p><strong>hi</strong></p>
```

has a **`p`** tag (nesting level 0—i.e., not nested in another tag) and a **`strong`** tag (nesting level 1).

13.7 Write a function that takes a list of dollar values separated by commas, converts each number from dollars to pounds (at an exchange rate 0.667 dollars per pound) and prints the results in a comma-separated list. Each converted value should have the £ symbol in front of it. This symbol can be obtained by passing the ASCII value of the symbol (**`156`**) to the **`chr`** function, which returns a string composed of that character. Ambitious programmers can attempt to do the conversion all in one statement.

13.8 Write a program that asks the user to enter a sentence and checks whether the sentence contains more than one space between words. If so, the program should remove the extra spaces. For example, **`"Hello    World"`** should be **`"Hello World"`**. (*Hint*: Use **`split`** and **`join`**.)

14

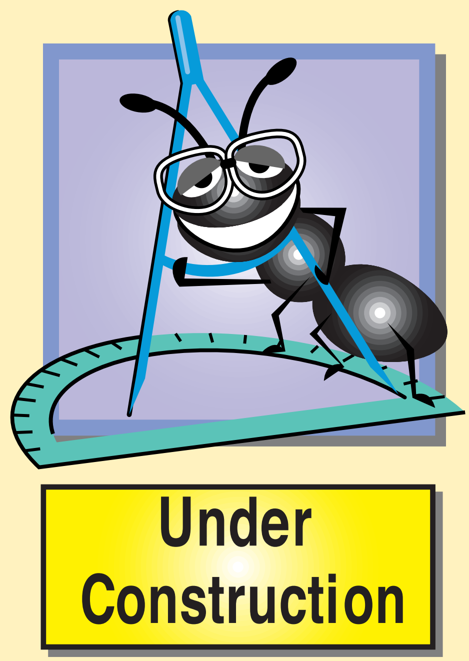
File Processing and Serialization

Objectives

- To create, read, write and update files.
- To become familiar with sequential-access file processing.
- To understand random-access file processing via module **`shelve`**.
- To specify high-performance, unformatted I/O operations.
- To understand the differences between formatted and raw data-file processing.
- To build a transaction-processing program with random-access file processing.
- To serialize complex objects for storage.

I read part of it all the way through.
Samuel Goldwyn

I can only assume that a "Do Not File" document is filed in a "Do Not File" file.
Senator Frank Church
Senate Intelligence Subcommittee Hearing, 1975



Outline

14.1 Introduction

Variables and sequences offer only temporary storage of data—the data is lost when a local variable "goes out of scope" or when the program terminates. By contrast, *files* are used for long-term retention of large amounts of data, even after the program that created the data terminates. Data maintained in files often is called *persistent* data. Computers store files on *secondary storage devices*, such as magnetic disks, optical disks and tapes. In this chapter, we explain how Python programs create, update and process data files. We consider both *sequential-access files* and *random-access files*, indicating the types of applications for which each is best suited. We compare formatted data-file processing and raw data-file processing, and we also examine various file-based data storage mechanisms, such as the **`shelve`** and **`cPickle`** modules.

14.2 Data Hierarchy

Ultimately, all data items processed by computers are reduced to combinations of zeros and ones. This occurs because it is simple and economical to build electronic devices that can assume two stable states—**`0`** represents one state and **`1`** represents the other. It is remarkable that the impressive functions performed by computers involve only the most fundamental manipulations of **`0`**s and **`1`**s.

The smallest data item that computers support is called a *bit* (short for *"binary digit"*—a digit that can assume one of two values). Each data item, or bit, can assume either the value **`0`** or the value **`1`**. Computer circuitry performs various bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing the value of a bit (from **`1`** to **`0`** or from **`0`** to **`1`**). For more information on binary numbers, refer to Appendix C, Number Systems.

Programming with data in the low-level form of bits is cumbersome. It is preferable to program with data in forms such as *decimal digits* (e.g., 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9), *letters*

(e.g., A through Z and a through z) and *special symbols* (e.g., $, @, %, &, *, (,), -, +, ", :, ?, /, etc.). Digits, letters and special symbols are referred to as *characters*. The set of all characters used to write programs and represent data items on a particular computer is called that computer's *character set*. Because computers can process only **1**s and **0**s, every character in a computer's character set is represented as a pattern of **1**s and **0**s. *Bytes* are composed of eight bits. Programmers create programs and data items with characters; computers manipulate and process these characters as patterns of bits.

Just as characters are composed of bits, *fields* are composed of characters (or bytes). A field is a group of characters that convey a meaning. For example, a field consisting of only uppercase and lowercase letters can represent a person's name.

Data items processed by computers form a *data hierarchy* in which data items become larger and more complex in structure in the progression from bits, to characters (bytes), to fields and upto larger data structures.

Typically, a *record*, which we can represent as a tuple, dictionary or instance in Python, is composed of several fields. In a payroll system, for example, a record for a particular employee might consist of the following fields:

1. Employee identification number
2. Name
3. Address
4. Hourly salary rate
5. Number of exemptions claimed
6. Year-to-date earnings
7. Amount of federal taxes withheld

Thus, a record is a group of related fields. In the preceding example, each field is associated with a particular employee. A company has a payroll record for each employee. A *file* is a group of related records.[1] A company's payroll file normally contains one record for each employee. Thus, a payroll file for a small company might contain only 22 records, whereas a payroll file for a large company might contain 100,000 records. It is not unusual for a company to have many files, some containing millions of characters of information. Figure 14.1 illustrates the data hierarchy.

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a *record key*. A record key identifies a record as belonging to a particular person or entity and distinguishes that record from all other records in the file. In the payroll record described previously, the employee-identification number would normally be chosen as the record key.

There are many ways to organize records in a file. The most common organization is called a *sequential file*, in which records typically are stored in order by the record-key field. In a payroll file, records usually are placed in order by employee-identification number. The first employee record in the file contains the lowest employee-identification number, and subsequent records contain increasingly higher employee-identification numbers.

---

1. Generally, a file can contain arbitrary data in arbitrary formats. In some operating systems, a file is viewed as nothing more than a collection of bytes. In such an operating system, any organization of bytes in a file (such as organizing the data into records) is a view created by the application's programmer.

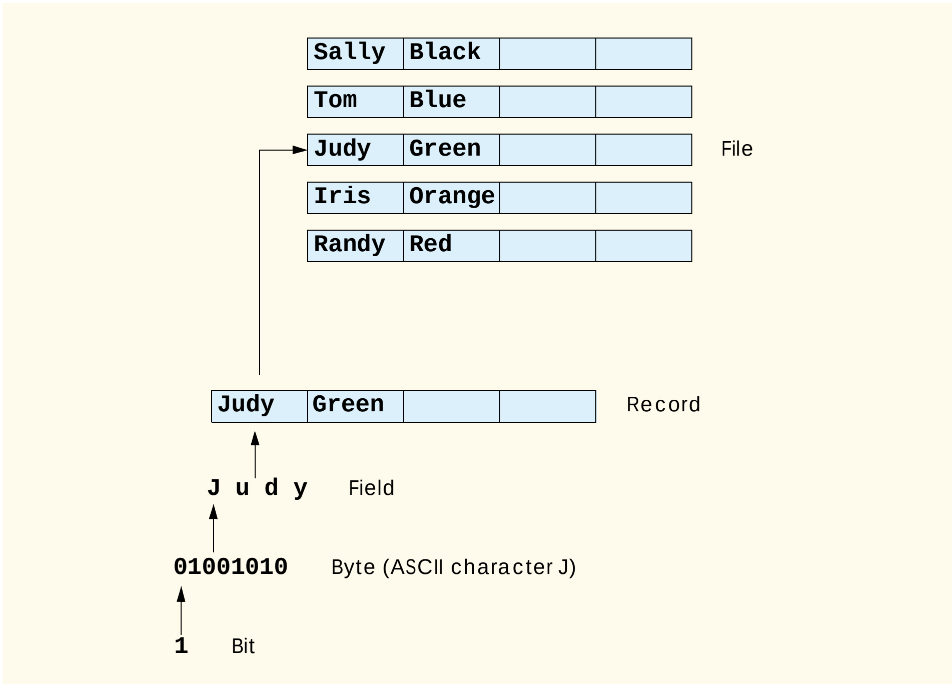


Fig. 14.1 Data hierarchy.

Most businesses use many different files to store data. For example, companies might have payroll files, accounts-receivable files (listing money due from clients), accounts-payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other types of files. Sometimes, a group of related files is called a *database*. A collection of programs designed to create and manage databases is called a *database management system (DBMS)*. We discuss databases in detail in Chapter 17, Database Application Programming Interface (DB-API).

14.3 Files and Streams

Python views each file as a sequential stream of bytes (Fig. 14.2). Each file ends either with an *end-of-file marker* or at a specific byte number recorded in a system-maintained administrative data structure. When a program *opens* a file, Python creates an object and associates a *stream* with that object.

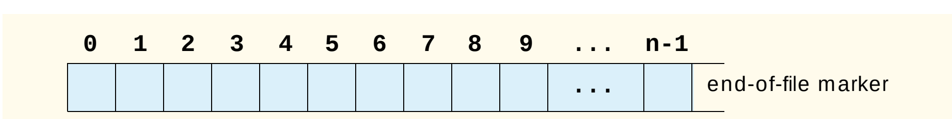


Fig. 14.2 Python's view of a file of ***n*** bytes.

When a Python program begins execution, Python creates three file streams—**`sys.stdin`** (*standard input stream*), **`sys.stdout`** (*standard output stream*) and **`sys.stderr`** (*standard error stream*). These streams provide communication channels between a program and a particular file or device. Python file streams are created regardless of whether a Python program imports the `sys` module, although a program must import the `sys` module to access the streams directly. Program input corresponds to `sys.stdin`. In fact, `raw_input` uses `sys.stdin` to retrieve user input. Program output corresponds to `sys.stdout`. The `print` statement sends information to the standard output stream, by default. Program errors are printed to `sys.stderr`.

The `sys.stdin` stream enables a program to receive input from the keyboard or other devices, the `sys.stdout` stream enables a program to output data to the screen or other devices and the `sys.stderr` stream enables a program to output error messages to the screen or other devices.

14.4 Creating a Sequential-Access File

Python imposes no structure on a file—notions like "records" do not exist in Python files. This means that the programmer must structure files to meet the requirements of applications. In the example in this section, we impose a record structure on a file.

Figure 14.3 creates a simple sequential-access file that might be used by an accounts-receivable system to track the money owed by a company's client. For each client, the program obtains an account number, the client's name and account balance (i.e., the amount the client owes the company). The data obtained for each client constitutes a record for that client. The account number represents the record key in this application; that is, the file will be created and maintained in account-number order. In our example, we assume a user enters the account information in account-number order. In a comprehensive accounts-receivable system, a sorting capability would be provided so the user could enter the records in any order—the records would then be sorted before being written to the file.

As stated previously, a programmer creates file-stream objects to open files. Function ***`open`***, which receives one required argument and two optional arguments, creates a stream object (line 8). The required argument for the new stream object is the *file name*; the two optional arguments are the *file-open mode* and the *buffering mode*.

```
1  # Fig. 14.3: fig14_03.py
2  # Opening and writing to a file.
3
4  import sys
5
6  # open file
7  try:
8     file = open( "clients.dat", "w" )  # open file in write mode
9  except IOError, message:              # file open failed
10    print >> sys.stderr, "File could not be opened:", message
11    sys.exit( 1 )
12
13 print "Enter the account, name and balance."
14 print "Enter end-of-file to end input."
```

Fig. 14.3 File-stream objects for opening and writing data to a file. (Part 1 of 2.)

```
while 1:

   try:
      accountLine = raw_input( "? " )   # get account entry
   except EOFError:
      break                             # user entered EOF
   else:
      print >> file, accountLine        # write entry to file

file.close()
```

```
Enter the account, name and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 14.3 File-stream objects for opening and writing data to a file. (Part 2 of 2.)

The file-open mode indicates whether a user can open the file for reading, writing or both. File-open mode **"w"** opens a file to output data to the file. Existing files opened with mode **"w"** are *truncated*—all data in the file is deleted—and re-created with the new data. If the specified file does not yet exist, then a file is created. The newly created file is assigned the name provided in the *file name* argument (i.e., **clients.dat**). If the location of the file is not specified in the file name argument, Python attempts to create the file in the current directory. If the file open-mode argument is not specified, the default value is **"r"**, which opens a file for reading. Figure 14.4 lists various file-open modes. The third argument to function **open**—the buffering-mode argument—is for advanced control of file input and output and usually is not specified. We do not assign a value to the buffering-mode argument in this example.

Mode	Description
"a"	Writes all output to the end of the file. If the indicated file does not exist, it is created.
"r"	Opens a file for input. If the file does not exist, an **IOError** exception is raised.
"r+"	Opens a file for input and output. If the file does not exist, causes an **IOError** exception.
"w"	Opens a file for output. If the file exists, it is truncated. If the file does not exist, one is created.
"w+"	Opens a file for input and output. If the file exists, it is truncated. If the file does not exist, one is created.

Fig. 14.4 File-open modes. (Part 1 of 2.)

Mode	Description
"ab", "rb", "r+b", "wb", "w+b"	Opens a file for binary (i.e., non-text) input or output. [*Note*: These modes are supported only on the Windows and Macintosh platforms.]

Fig. 14.4 File-open modes. (Part 2 of 2.)

Common Programming Error 14.1

Opening an existing file for output (`"w"`) when the user wants to preserve the file is a logic error, because the contents of the file are discarded without warning.

When **open** encounters an error, the function raises an ***IOError*** exception. Some possible errors include attempting to open a file for reading that does not exist, opening a read-only file for writing and opening a file for writing when no disk space is available.

In Fig. 14.3, if **open** raises an **IOError** exception, line 10 prints the error message **"File could not be opened"** to **sys.stderr**. By default, the **print** statement sends output to the **sys.stdout** file object. Programs can *redirect* output from the **print** statement to print to a different file object. In our example, the statement

```
print >> sys.stderr, "File could not be opened:", message
```

redirects output to the **sys.stderr** (standard error) file object. When **>>** symbol follows the **print** keyword, the **print** statement redirects the output to the file object that appears to the right of **>>**. A comma follows the output file object, and the value to print follows the comma.

Common Programming Error 14.2

When redirecting file output with **>>***, forgetting to put a comma (***,***) after the file object is a syntax error.*

Redirecting output with **>>** was added to Python in version 2.0. For earlier versions, or to support multiple versions, the effect of redirecting output with the **>>** symbol can be accomplished with file object method **write** as follows:

```
sys.stderr.write( output )
```

The table in Fig. 14.5 lists the file-object methods.

Method	Description
close()	Closes the file object.
fileno()	Returns an integer that is the file's *file descriptor* (i.e., the number the operating system uses to maintain information about the file).

Fig. 14.5 File-object methods. (Part 1 of 2.)

Method	Description
`flush()`	Flushes the file's *buffer*. A buffer contains the information to be written to or read from a file. Flushing the buffer performs the read or write operation.
`isatty()`	Returns **`1`** if the file object is a *tty* (terminal) device.
`read(` *[size]* **`)`**	Reads data from the file. If *size* is not specified, the method reads all data to the end of the file. If argument *size* is specified, the method reads at most *size* bytes from the file.
`readline(` *[size]* **`)`**	Reads one line from the file. If *size* is not specified, the method reads to the end of the line. If *size* is specified, the method reads at most *size* bytes from the line.
`readlines(` *[size]* **`)`**	Reads lines from the file and returns the lines in a list. If *size* is not specified, the method reads to the end of the file. If *size* is specified, the method reads at most *size* bytes.
`seek(` *offset[, location]* **`)`**	Moves the file position *offset* bytes. If *location* is not specified, the file position moves *offset* bytes from the beginning of the file. If *location* is specified, the file position moves *offset* bytes from *location*. Section 14.5 discusses **`seek`** in detail.
`tell()`	Returns the file's current position.
`truncate(` *[size]* **`)`**	Truncates data in the file. If *size* is not specified, all data is deleted. If *size* is specified, the file is truncated to contain at most *size* bytes.
`write(` *output* **`)`**	Writes the string *output* to the file.
`writelines(` *outputList* **`)`**	Writes each string in *outputList* to the file.
`xreadlines()`	Similar to **`readlines`**, except the method implements a more memory-efficient way to read a file. The method returns an **`xreadlines`** object that a program can iterate over to retrieve the information. See **`www.python.org/doc/current/lib/module-xreadlines.html`** for information on **`xreadlines`** objects.

Fig. 14.5 File-object methods. (Part 2 of 2.)

If an error occurs when the program in Fig. 14.3 opens a file, function **`sys.exit`** (line 11) terminates the program. Function **`sys.exit`** returns its optional argument to the environment from which the program was invoked. Argument **`0`** (the default) indicates normal program termination; any other value indicates that the program terminated due to an error. The calling environment (most likely the operating system) uses the value returned by **`sys.exit`** to respond to the error appropriately.

If the file, **`clients.dat`**, opens successfully, the program processes data. Lines 13–14 prompt the user to enter the various fields for each record, or the end-of-file marker when data entry is completed.

Lines 16–23 extract each set of data from the standard input using a **`try`**/**`except`**/**`else`** block in a repetition structure. Function **`raw_input`** retrieves a line of input from

the user. If the user enters the end-of-file character, **raw_input** raises an ***EOFError*** exception. Lines 20–21 catch this error and use a **break** statement to exit the infinite **while** loop. If the user does not enter the end-of-file character, the **else** block (lines 22–23) executes and prints the user-entered line to the output file using the **>>** symbol.

The ***close*** method (line 25) closes the file object after the **while** loop terminates. Although Python closes open files when a program terminates, it is good practice to close a file with the **close** method as soon as the program no longer needs the file.

Performance Tip 14.1

Close each file explicitly as soon as it is known that the program will not reference the file again. This can reduce resource use in a program that continues executing after it no longer needs a particular file. This practice also improves program clarity.

In the sample execution for the program of Fig. 14.3, the user enters information for five accounts and signals that data entry is complete by entering end-of-file. This dialog does not show how the data records actually appear in the file. To verify that the file has been created successfully, the next section demonstrates a program that reads the file and displays its contents.

14.5 Reading Data from a Sequential-Access File

Data is stored in files so that the data can be retrieved for processing at a later time. The previous section demonstrated how to create a sequential-access file. In this section, we discuss how to read (or retrieve) data sequentially from a file.

Figure 14.6 reads records from the file **clients.dat** created by the program of Fig. 14.3 and displays the contents of each record. Files are opened for reading by passing an **"r"** as the second argument to function **open** (line 8). If the locatin of the file is not specified in the file name argument to **open**, Python attempts to locate the file in the current directory.

```
1  # Fig. 14.6: fig14_06.py
2  # Reading and printing a file.
3
4  import sys
5
6  # open file
7  try:
8     file = open( "clients.dat", "r" )
9  except IOError:
10    print >> sys.stderr, "File could not be opened"
11    sys.exit( 1 )
12
13 records = file.readlines()   # retrieve list of lines in file
14
15 print "Account".ljust( 10 ),
16 print "Name".ljust( 10 ),
17 print "Balance".rjust( 10 )
18
```

Fig. 14.6 Data read from a sequential-access file. (Part 1 of 2.)

```
19  for record in records:          # format each line
20     fields = record.split()
21     print fields[ 0 ].ljust( 10 ),
22     print fields[ 1 ].ljust( 10 ),
23     print fields[ 2 ].rjust( 10 )
24
25  file.close()
```

```
Account    Name           Balance
100        Jones            24.98
200        Doe             345.67
300        White             0.00
400        Stone           -42.16
500        Rich            224.62
```

Fig. 14.6 Data read from a sequential-access file. (Part 2 of 2.)

File objects are opened for reading by default, so the statement

```
file = open( "clients.dat" )
```

also opens **clients.dat** for reading.

Good Programming Practice 14.1

A programmer should set a file to open for reading only (using **"r"***) if the contents of the file should not be modified. This prevents unintentional modifications of a file's contents. This is an example of the principle of least privilege.*

Method ***readlines*** (line 13) reads the entire file contents of Fig. 14.3 into the program. This method returns a list of the lines in the file, which the program stores in variable **records**. For each line (**record**) in the file, method **split** returns the words (**fields**) in the line as a list. Lines 19–23 output the fields. Methods **ljust** and **rjust** left- and right-justify the fields, respectively, to format the output. Method **close** (line 25) closes the file associated with the file object.

Python version 2.2 contains additional features that enable the programmer to use a file object in a **for** statement. For example, line 19 in the above example could be replaced by:

```
for record in file:
```

in a program that uses Python 2.2. This technique reads one line of **file** at a time and assigns the line to **record**. The program can process that line immediately. Iterating over the lines in a file in this manner can be more efficient than reading the contents of a large file with method **readlines**, which requires the program to wait for the entire file to be read into memory before any of the file's contents can be processed.

To retrieve data sequentially from a file, programs normally start from the beginning of the file and read all the data consecutively until the desired data is found. It sometimes is necessary to process a file sequentially several times (from the beginning of the file) during the execution of a program. File objects provide method ***seek*** for repositioning the *file-position pointer* (which contains the byte number of the next byte to be read from or written to the file). The statement

```
file.seek( 0, 0 )
```

repositions the file-position pointer at the beginning of the file. The first argument **seek** takes is the *offset*, which is an integer value that specifies the location in the file as a number of bytes from the *seek direction* of the file. The second (optional) argument is the seek direction, or *location*, from which the offset begins. The seek direction can be **0** (the default) for positioning relative to the beginning of a file, **1** for positioning relative to the current position in a file or **2** for positioning relative to the end of a file. Some examples of positioning the file position pointer are

```
# position to the nth byte of file
# assumes seek direction is 0
file.seek( n )

# position n bytes forward in file
file.seek( n, 1 )

# position n bytes backward from end of file
file.seek( -n, 2 )

# position at end of file
file.seek( 0, 2 )
```

File-object method ***tell*** returns the current location of the file-position pointer. The following statement assigns the current file-position pointer value to variable **location**

```
location = file.tell()
```

Figure 14.7 uses **seek** in a program that enables a credit manager to display the account information for customers with zero balances (i.e., customers who do not owe any money), credit balances (i.e., customers to whom the company owes money) and debit balances (i.e., customers who owe the company money for goods and services received in the past). The program displays a menu and allows the credit manager to enter one of three options to obtain credit information. Option 1 produces a list of accounts with zero balances (lines 56–57). Option 2 produces a list of accounts with credit balances (lines 58–59). Option 3 produces a list of accounts with debit balances (lines 60–61). Option 4 terminates the program execution (lines 62–63). Entering an invalid option causes the program to prompt the user to enter another choice (lines 64–65).

Lines 52–77 process the request for each request that is not option 4. Method **readline** (line 67) reads one line from the file and moves the file-position pointer to the next line in the file. When method **readline** has finished reading all lines from the file (i.e., the program has reached the end of the file), **readline** returns the empty string (**""**).

Method **split** (line 71) unpacks each record to three variables—**account**, **name** and **balance**. The program calls function **shouldDisplay** (lines 18–29), which returns **1** (true), if a record should be displayed. If applicable, function **outputLine** (lines 32–36) displays the record fields.

```
1  # Fig. 14.7: fig14_07.py
2  # Credit inquiry program.
3
```

Fig. 14.7 Credit-inquiry program. (Part 1 of 3.)

```
import sys

# retrieve one user command
def getRequest():

   while 1:
      request = int( raw_input( "\n? " ) )

      if 1 <= request <= 4:
         break

   return request

# determine if balance should be displayed, based on type
def shouldDisplay( accountType, balance ):

   if accountType == 2 and balance < 0:     # credit balance
      return 1

   elif accountType == 3 and balance > 0:   # debit balance
      return 1

   elif accountType == 1 and balance == 0:  # zero balance
      return 1

   else: return 0

# print formatted balance data
def outputLine( account, name, balance ):

   print account.ljust( 10 ),
   print name.ljust( 10 ),
   print balance.rjust( 10 )

# open file
try:
   file = open( "clients.dat", "r" )
except IOError:
   print >> sys.stderr, "File could not be opened"
   sys.exit( 1 )

print "Enter request"
print "1 - List accounts with zero balances"
print "2 - List accounts with credit balances"
print "3 - List accounts with debit balances"
print "4 - End of run"

# process user request(s)
while 1:

   request = getRequest()   # get user request

```

Fig. 14.7 Credit-inquiry program. (Part 2 of 3.)

```
      if request == 1:         # zero balances
         print "\nAccounts with zero balances:"
      elif request == 2:       # credit balances
         print "\nAccounts with credit balances:"
      elif request == 3:       # debit balances
         print "\nAccounts with debit balances:"
      elif request == 4:       # exit loop
         break
      else:  # getRequest should never let program reach here
         print "\nInvalid request."

      currentRecord = file.readline()    # get first record

      # process each line
      while ( currentRecord != "" ):
         account, name, balance = currentRecord.split()
         balance = float( balance )

         if shouldDisplay( request, balance ):
            outputLine( account, name, str( balance ) )

         currentRecord = file.readline() # get next record

      file.seek( 0, 0 )                  # move to beginning of file

   print "\nEnd of run."
   file.close()                          # close file
```

```
Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run

? 1

Accounts with zero balances:
300        White             0.0

? 2

Accounts with credit balances:
400        Stone          -42.16

? 3

Accounts with debit balances:
100        Jones           24.98
200        Doe            345.67
500        Rich           224.62

? 4

End of run.
```

Fig. 14.7 Credit-inquiry program. (Part 3 of 3.)

14.6 Updating Sequential-Access Files

Data that is formatted and written to a sequential-access file (as shown in the previous sections) cannot be modified without the risk of destroying other data in the file. For example, if the name "**White**" needs to be changed to "**Williams**," the old name cannot simply be overwritten. In Fig. 14.7, the record for **White** was written to the file as

```
300 White 0.00
```

If a user overwrites this record with the name "**Williams**," the record appears as

```
300 Williams00
```

The new last name contains three more characters than the original last name, so the characters beyond the second "**i**" in "**Williams**" overwrite the other characters in the line. The problem here is that in the formatted input/output model, fields (records) can vary in size. For example, 7, 14, –117, 2074 and 27383 are all integers and are stored in the same number of "raw data" bytes internally, but when these integers are output as formatted text (character sequences) to the screen or to a file, they become different-sized fields. Therefore, the formatted input/output model usually is not used to update records in place.

Updating data in sequential-access files is possible, but it is awkward. For example, to make the preceding name change, the records before **300 White 0.00** in a sequential-access file could be copied to a separate file, the updated record could then be written to this file and the records after **300 White 0.00** could be copied to this file. However, this can be cumbersome, because it requires processing every record in the file to update one record. If many records are being updated in one pass of the file, then the effort this technique requires would be unacceptable.

14.7 Random-Access Files

We have explained how to create sequential-access files and how to search through such files to locate particular information. However, sequential-access files are inappropriate for so-called *"instant-access" applications*, in which a particular record of information must be located immediately. Popular instant-access applications include airline reservation systems, banking systems, point-of-sale systems, automated-teller machines (ATMs) and other kinds of *transaction-processing systems* that require rapid access to specific data. The bank at which an individual keeps an account may have hundreds of thousands or even millions of other customers; however, when that individual uses an ATM, the appropriate account is checked for sufficient funds in seconds. Instant access is possible with *random-access files* (sometimes called *direct-access files*). Individual records of a random-access file can be accessed directly (and quickly) without searching through other records.

As we discussed earlier in this chapter, Python does not impose structure on files, so applications that use random-access files must create the random-access capability. There are a variety of techniques for creating random-access files. Perhaps the simplest involves requiring that all records in a file be of uniform-fixed length. The use of fixed-length records enable a program to calculate (as a function of the record size and the record key) the exact location of any record in relation to the beginning of the file. We soon demonstrate how this facilitates immediate access to specific records, even in large files.

Figure 14.8 presents a view of a random-access file composed of fixed-length records Each record is 100 bytes long. A random-access file is like a railroad train with many

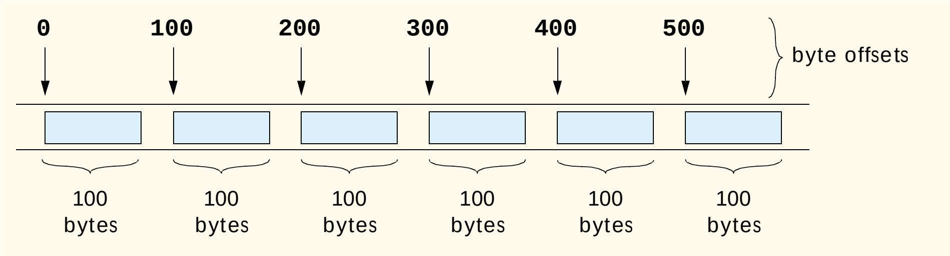


Fig. 14.8 Structure of a random-access file.

cars—some empty, some with contents. Data can be inserted into a random-access file without destroying other data in the file. In addition, previously stored data can be updated or deleted without rewriting the entire file. In the following sections, we explain how to create a random-access file, enter data to that file, read the data, update the data and delete data that is no longer needed.

14.8 Simulating a Random-Access File: The `shelve` Module

Random-access file-processing programs rarely write a single field to a file. Often, these programs write one record (or object) at a time. Random-access files can be created in other programming languages by defining a class that represents the record to be written to the file. In such programming languages, program that uses the class then reads and writes instances to a random-access file based on the size of the class (i.e., the number of bytes an instance of the class occupies). Python provides module ***`shelve`*** to simulate such behavior, so a programmer does not need to write a new class. We use this module in the following examples.

Consider the following problem statement for a credit-processing application:

> *Create a transaction-processing program capable of storing a maximum of 100 fixed-length records for a company that can have a maximum of 100 customers. Each record consists of an account number (that acts as a record key), a last name, a first name and a balance. The program can update an account, insert a new account, delete an account and list all the account records in a file.*

The next several sections introduce the techniques necessary to create this credit-processing program. We can use module **`shelve`** to read and write records in a file. To accomplish this, we create **`shelve`** objects to represent the records. These objects have a dictionary interface—the record key accesses a record's information. In our example, the record key is the account number, and the record value is a list that contains the customer's last name, first name and account balance.

14.9 Writing Data to a `shelve` File

Figure 14.9 retrieves account information from the user and writes the data to the **`shelve`** file **`credit.dat`**. Line 9 opens the **`shelve`** file **`credit.dat`** using function **`shelve.open`**. This function resembles the Python function **`open`** used for opening regular files. If the file does not exist, **`shelve`** function **`open`** creates the file. If an error occurs when opening the file, the function raises an **`IOError`** exception.

```python
# Fig. 14.9: fig14_09.py
# Writing to shelve file.

import sys
import shelve

# open shelve file
try:
   outCredit = shelve.open( "credit.dat" )
except IOError:
   print >> sys.stderr, "File could not be opened"
   sys.exit( 1 )

print "Enter account number (1 to 100, 0 to end input)"

# get account information
while 1:

   # get account information
   accountNumber = int( raw_input(
         "\nEnter account number\n? " ) )

   if 0 < accountNumber <= 100:

      print "Enter lastname, firstname, balance"
      currentData = raw_input( "? " )

      outCredit[ str( accountNumber ) ] = currentData.split()

   elif accountNumber == 0:
      break

outCredit.close()   # close shelve file
```

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00

Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54

Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98

Enter account number
? 88
```

(continued on next page)

Fig. 14.9 Data written to a `shelve` file. (Part 1 of 2.)

(continued from previous page)

```
Enter lastname, firstname, balance
? Smith Dave 258.34

Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33

Enter account number
? 0
```

Fig. 14.9 Data written to a **`shelve`** file. (Part 2 of 2.)

Lines 20–21 prompt the user for the account numbers in the range 1–100, inclusive. Line 28 writes data to the **`shelve`** file. The program manipulates the data in a **`shelve`** file through a dictionary interface. Each key in a **`shelve`** file must be a string; therefore, function **`str`** converts the integer value **`accountNumber`** to a string (line 28). Method **`split`** converts the user-entered data into a list, which is stored as the record key's value (line 28). When the user enters **`0`** to indicate the end of data, the file object's **`close`** method closes the **`shelve`** file (line 33).

14.10 Retrieving Data from a `shelve` File

In the previous section, we created a **`shelve`** file and wrote data to that file. In this section, we present a program (Fig. 14.10) that iterates through the file and prints each record.

```
# Fig. 14.10: fig14_10.py
# Reading shelve file.

import sys
import shelve

# print formatted credit data
def outputLine( account, aList ):

   print account.ljust( 10 ),
   print aList[ 0 ].ljust( 10 ),
   print aList[ 1 ].ljust( 10 ),
   print aList[ 2 ].rjust( 10 )

# open shelve file
try:
   creditFile = shelve.open( "credit.dat" )
except IOError:
   print >> sys.stderr, "File could not be opened"
   sys.exit( 1 )

print "Account".ljust( 10 ),
```

Fig. 14.10 Data read from a **`shelve`** file. (Part 1 of 2.)

```
23   print "Last Name".ljust( 10 ),
24   print "First Name".ljust( 10 ),
25   print "Balance".rjust( 10 )
26
27   # display each account
28   for accountNumber in creditFile.keys():
29      outputLine( accountNumber, creditFile[ accountNumber ] )
30
31   creditFile.close()   # close shelve file
```

```
Account     Last Name  First Name    Balance
37          Barker     Doug             0.00
88          Smith      Dave           258.34
33          Dunn       Stacey         314.33
29          Brown      Nancy          -24.54
96          Stone      Sam             34.98
```

Fig. 14.10 Data read from a **shelve** file. (Part 2 of 2.)

Method ***keys*** returns a list of the record keys in the **shelve** file (line 28). A **for** loop iterates over this list and passes each record key and its value to function **outputLine**. Function **outputLine** (lines 8–13) prints the record key and its associated values.

14.11 Example: A Transaction-Processing Program

We now develop a substantial transaction-processing program (Fig. 14.11) that uses a **shelve** file to achieve "instant-access" processing. The program maintains a bank's account information. Users of this program can update existing accounts, add new accounts, delete accounts and store formatted listings of all the current accounts in text files.

```
# Fig. 14.11: fig14_11.py
# Reads shelve file, updates data
# already written to file, creates data
# to be placed in file and deletes data
# already in file.

import sys
import shelve

# prompt for input menu choice
def enterChoice():

   print "\nEnter your choice"
   print "1 - store a formatted text file of accounts"
   print "    called \"print.txt\" for printing"
   print "2 - update an account"
   print "3 - add a new account"
   print "4 - delete an account"
   print "5 - end program"
```

Fig. 14.11 Bank-account program. (Part 1 of 4.)

```

   while 1:
      menuChoice = int( raw_input( "? " ) )

      if not 1 <= menuChoice <= 5:
         print >> sys.stderr, "Incorrect choice"

      else:
         break

   return menuChoice

# create formatted text file for printing
def textFile( readFromFile ):

   # open text file
   try:
      outputFile = open( "print.txt", "w" )
   except IOError:
      print >> sys.stderr, "File could not be opened."
      sys.exit( 1 )

   print >> outputFile, "Account".ljust( 10 ),
   print >> outputFile, "Last Name".ljust( 10 ),
   print >> outputFile, "First Name".ljust( 10 ),
   print >> outputFile, "Balance".rjust( 10 )

   # print shelve values to text file
   for key in readFromFile.keys():
      print >> outputFile, key.ljust( 10 ),
      print >> outputFile, readFromFile[ key ][ 0 ].ljust( 10 ),
      print >> outputFile, readFromFile[ key ][ 1 ].ljust( 10 ),
      print >> outputFile, readFromFile[ key ][ 2 ].rjust( 10 )

   outputFile.close()

# update account balance
def updateRecord( updateFile ):

   account = getAccount( "Enter account to update" )

   if updateFile.has_key( account ):
      outputLine( account, updateFile[ account ] ) # get record

      transaction = raw_input(
         "\nEnter charge (+) or payment (-): " )

      # create temporary record to alter data
      tempRecord = updateFile[ account ]
      tempBalance = float( tempRecord[ 2 ] )
      tempBalance += float( transaction )
      tempBalance = "%.2f" % tempBalance
      tempRecord[ 2 ] = tempBalance
```

Fig. 14.11 Bank-account program. (Part 2 of 4.)

```

      # update record in shelve
      del updateFile[ account ]   # remove old record first
      updateFile[ account ] = tempRecord
      outputLine( account, updateFile[ account ] )
   else:
      print >> sys.stderr, "Account #", account, \
         "does not exist."

# create and insert new record
def newRecord( insertInFile ):

   account = getAccount( "Enter new account number" )

   if not insertInFile.has_key( account ):
      print "Enter lastname, firstname, balance"
      currentData = raw_input( "? " )
      insertInFile[ account ] = currentData.split()
   else:
      print >> sys.stderr, "Account #", account, "exists."

# delete existing record
def deleteRecord( deleteFromFile ):

   account = getAccount( "Enter account to delete" )

   if deleteFromFile.has_key( account ):
      del deleteFromFile[ account ]
      print "Account #", account, "deleted."
   else:
      print >> sys.stderr, "Account #", account, \
         "does not exist."


# output line of client information
def outputLine( account, record ):

   print account.ljust( 10 ),
   print record[ 0 ].ljust( 10 ),
   print record[ 1 ].ljust( 10 ),
   print record[ 2 ].rjust( 10 )

# get account number from keyboard
def getAccount( prompt ):

   while 1:
      account = raw_input( prompt + " (1 - 100): " )

      if 1 <= int( account ) <= 100:
         break

   return account

```

Fig. 14.11 Bank-account program. (Part 3 of 4.)

```
126 # list of functions that correspond to user options
127 options = [ textFile, updateRecord, newRecord, deleteRecord ]
128 
129 # open shelve file
130 try:
131    creditFile = shelve.open( "credit.dat" )
132 except IOError:
133    print >> sys.stderr, "File could not be opened."
134    sys.exit( 1 )
135 
136 # process user commands
137 while 1:
138 
139    choice = enterChoice()              # get user menu choice
140 
141    if choice == 5:
142       break
143 
144    options[ choice - 1 ]( creditFile )  # invoke option function
145 
146 creditFile.close()                     # close shelve file
```

Fig. 14.11 Bank-account program. (Part 4 of 4.)

Execute the program in Fig. 14.9 to insert data in the file that is used in this transaction-processing program (Fig. 14.11). The transaction-processing program offers a user five options (**1**–**5**) with which to work in the program. Option **1** calls function **textFile** (lines 33–54), which stores a formatted list of all the account information in a text file called **print.txt**. From this file, a user can print a list of account information. Function **textFile** takes a **shelve** file as an argument and uses the data in that **shelve** file to create the text file. Function **outputLine** (lines 108–113) outputs the data to file **stdout**. After a user chooses option **1**, the file **print.txt** contains the following text:

```
Account     Last Name   First Name     Balance
37          Barker      Doug              0.00
88          Smith       Dave            258.54
33          Dunn        Stacey          314.33
29          Brown       Nancy           -24.54
96          Stone       Sam              34.98
```

When a user selects option **2**, the program calls function **updateRecord** (lines 57–80) to update an account. First, the function determines whether the record that the user specifies exists, because the function can update only existing records. If the record exists, it is read into variable **tempRecord**. Lines 69–70 convert the string representation of the account balance to a floating-point value before manipulating its numerical value. Before updating a record in the **shelve** file, the program must delete the existing record for the specified account; keyword **del** (line 75) deletes the current record. Line 76 updates the record by assigning the new record values to the corresponding account number (record key). The program then outputs the updated values. The following is a typical output for this option:

```
Enter account to update (1 - 100): 37
37          Barker     Doug              0.00

Enter charge (+) or payment (-): +87.99
37          Barker     Doug             87.99
```

Portability Tip 14.1

Not all Python platforms require the value of a record to be deleted from a **`shelve`** *file before updating that record. However, using* **`del`** *to delete a record value before updating it, ensures that the update occurs properly across Python platforms.*

Option **`3`** calls function **`newRecord`** to enable a user to add a new account. This function adds an account in the same manner as that of the program of Fig. 14.9. If the user enters an account number for an existing account, **`newRecord`** displays a message that the account exists and the program allows the user to select the next operation to perform. A typical output for option **`3`** is as follows:

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

Option **`4`** calls function **`deleteRecord`** to remove a record that is no longer needed. The program prompts the user to enter an account number. If the account number exists, the program uses keyword **`del`** to delete that record from the **`shelve`** file, then displays a message to inform the user that the record has been deleted. However, if the account number does not exist, the program displays an error message. A typical output for option **`4`** is as follows:

```
Enter account to delete (1 - 100): 29
Account # 29 deleted.
```

Option **`5`** terminates the program. The main portion of the program (lines 127–146) creates a list of functions that correspond to the user-menu options (line 127). The program then opens the **`shelve`** file for the bank accounts and gets the user's menu choice.

Line 144 calls a function that corresponds with a user option. Recall that parentheses (**`()`**) are Python operators. When used in conjunction with the function name (e.g., **`textFile`**), the operator calls the function and passes any indicated arguments. Variable **`options`** holds a list of function names, so a statement such as

```
options[ 0 ]( creditFile )
```

invokes function **`textFile`** (the first function in the list) and passes **`creditFile`** as an argument. Statements like this avoid the need for long **`if`**/**`else`** statements that determine the user menu option and call the appropriate function.

14.12 Object Serialization

Serialization, or *pickling*, converts complex object types, such as user-defined classes, to sets of bytes for storage or for transmission over a network. Pickling also is referred to as *flattening* or *marshalling*. Python provides both modules ***pickle*** and ***cPickle*** to perform pickling. In this text we use **cPickle**, a module written in C, instead of the Python module **pickle**. We choose to use **cPickle**, because modules written in compiled languages, such as C, execute faster than do interpreted languages such as Python. Figure 14.12 demonstrates pickling and storing a list in a file.

Performance Tip 14.2

*Module **cPickle** executes more efficiently than does module **pickle**, because **cPickle** is implemented in C and compiled into native machine language on each platform.*

```
# Fig. 14.12: fig14_12.py
# Opening and writing pickled object to file.

import sys, cPickle

# open file
try:
   file = open( "users.dat", "w" )     # open file in write mode
except IOError, message:               # file open failed
   print >> sys.stderr, "File could not be opened:", message
   sys.exit( 1 )

print "Enter the user name, name and date of birth."
print "Enter end-of-file to end input."

inputList = []

while 1:

   try:
      accountLine = raw_input( "? " )         # get user entry
   except EOFError:
      break                                   # user-entered EOF
   else:
      inputList.append( accountLine.split() ) # append entry

cPickle.dump( inputList, file )  # write pickled object to file

file.close()
```

```
Enter the user name, name and date of birth.
Enter end-of-file to end input.
? mike Michael 4/3/60
? joe Joseph 12/5/71
? amy Amelia 7/10/80
? jan Janice 8/18/74
? ^Z
```

Fig. 14.12 Pickled object written to a file.

Line 8 opens **user.dat**, the file in which the pickled object resides. Variable **inputList**, initialized in line 16, is a list that contains the user-entered information to pickle. Lines 18–25 prompt the user to enter information and append the user's entries to **inputList**. Function ***cPickle.dump*** (line 27) pickles **inputList** to the file. The first argument to **dump** is the object to pickle and the second argument is the file object that represents the file in which method **dump** will store the pickled object. The function converts **inputList** to a series of bytes and writes the stream to the file. Line 29 calls file object method **close** to close the file.

A program can convert pickled data back to the original format by *unpickling* the data. Figure 14.13 demonstrates unpickling. This example uses the pickled file created by the program in Fig. 14.12.

The program first opens file **users.dat** (lines 7–11). Function ***cPickle.load*** (line 13) unpickles the data in the file. The function takes as an argument a file object that contains a pickled object, converts the pickled object into a Python object and returns a reference to the unpickled object. We assign this reference to variable **records**. The program then closes the file, because the file is no longer needed (line 14). The remainder of the program (lines 16–23) displays the unpickled data by iterating over the list of lists.

```
1  # Fig. 14.13: fig14_13.py
2  # Reading and printing pickled object in a file.
3
4  import sys, cPickle
5
6  # open file
7  try:
8     file = open( "users.dat", "r" )
9  except IOError:
10    print >> sys.stderr, "File could not be opened"
11    sys.exit( 1 )
12
13 records = cPickle.load( file )  # retrieve list of lines in file
14 file.close()
15
16 print "Username".ljust( 15 ),
17 print "Name".ljust( 10 ),
18 print "Date of birth".rjust( 20 )
19
20 for record in records:          # format each line
21    print record[ 0 ].ljust( 15 ),
22    print record[ 1 ].ljust( 10 ),
23    print record[ 2 ].rjust( 20 )
```

```
Username        Name               Date of birth
mike            Michael                   4/3/60
joe             Joseph                   12/5/71
amy             Amelia                   7/10/80
jan             Janice                   8/18/74
```

Fig. 14.13 Pickled object read from a file.

SUMMARY

- Files are used for long-term retention of large amounts of data.
- Computers store files on secondary storage devices, such as magnetic disks, optical disks and tapes.
- Ultimately, all data items processed by digital computers are reduced to combinations of zeros and ones. This occurs because it is simple and economical to build electronic devices that can assume two stable states—**0** represents one state, and **1** represents the other.
- The smallest data item that computers support is called a bit (short for "binary digit"—a digit that can assume one of two values). Each data item, or bit can assume either the value **0** or the value **1**.
- It is preferable to program with data forms such as decimal digits (e.g., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9), letters (e.g., A through Z and a through z) and special symbols (e.g., $, @, %, &, *, (,), -, +, ", :, ?, /, etc.).
- Digits, letters and special symbols are referred to as characters.
- The set of all characters used to write programs and represent data items on a particular computer is called that computer's character set.
- Because computers can process only **1**s and **0**s, every character in a computer's character set is represented as a sequence of **1**s and **0**s (called a byte). Bytes are composed of eight bits.
- Just as characters are composed of bits, fields are composed of characters (or bytes). A field is a group of characters that convey a meaning.
- Data items processed by computers form a data hierarchy in which data items become larger and more complex in structure in the progression from bits, to characters (bytes), to fields and up to larger data structures.
- A record, which we can implement as a tuple, a dictionary or instance in Python, is a group of related fields.
- To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key. A record key identifies a record as belonging to a particular person or entity and distinguishes that record from all other records in the file.
- There are many ways to organize records in a file. In the most common organization is a sequential file, in which records typically are stored in order by the record-key field.
- Sometimes, a group of related files is called a database.
- A collection of programs designed to create and manage databases is called a database management system (DBMS).
- Python views each file as a sequential stream of bytes.
- Python imposes no structure on a file—notions like "records" do not exist in Python files.
- Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained administrative data structure.
- When a file is opened, Python creates an object and associates a stream with that object.
- Python creates three file streams—**sys.stdin** (standard input stream), **sys.stdout** (standard output stream) and **sys.stderr** (standard error stream). These streams provide communication channels between a program and a particular file or device.
- A program must import the **sys** module to access the three file streams directly.
- Program input corresponds to **sys.stdin**. Function **raw_input** uses **sys.stdin** to get input from the user.
- Program output corresponds with **sys.stdout**. By default, the **print** statement sends information to the standard output stream.

- Program errors are printed to **sys.stderr**.
- Function **open**, which takes one required argument and two optional arguments, creates a new stream objects. The required argument, for the new stream object, is the file name; the two optional arguments are the file open mode and the buffering mode.
- The file open mode specifies whether the file should be opened for reading, writing or both.
- The file open mode **"w"** opens a file to output data to the file; the file open mode **"w+"** opens a file to append data to the end of the file (without modifying any data already in the file).
- Existing files opened with mode **"w"** are truncated—all data in the file is deleted. If the specified file does not yet exist, then a file is created. The newly created file is assigned the name provided in the file-name argument.
- The default file-open mode is **"r"**, which opens a file for reading.
- The third argument to function open—the buffering-mode argument—is for advanced control of file input and output and usually is not specified.
- When **open** encounters an error, the function raises an **IOError** exception.
- Programs can redirect output from the **print** statement to print to a different file object.
- When the **>>** symbol follows the **print**, **print** redirects output to the file object that appears to the right of **>>**. A comma follows the output file object; the value to be printed follows the comma.
- Function **sys.exit** terminates a program and returns its optional argument to the environment from which the program was invoked. Argument value **0** (the default) indicates normal program termination. Any other value indicates that the program terminated due to an error. The calling environment can use the value returned by **sys.exit** to respond appropriately to the error.
- Invoking methods **read**, **readline**, **readlines** and **xreadlines** on the end-of-file character raises an **EOFError** exception.
- Method **close** closes a file object. Although Python closes open files when a program terminates, it is good practice to close a file explicitly as soon as the program no longer needs the file.
- Data is stored in files so that they can be retrieved for processing.
- Method **readlines** reads an entire file into a program.
- Methods **ljust** and **rjust** left-and right-justify fields, respectively.
- To retrieve data sequentially from a file, programs normally start reading from the beginning of the file, and read all the data consecutively until the desired data is found.
- Method **seek** repositions the file position pointer. The first argument to **seek** is an offset, which is an integer value that specifies the location in the file as a number of bytes from the seek direction. The second (optional) argument is a seek direction, or location, from which the offset begins.
- File-object method **tell** returns the current location of the file-position pointer.
- Method **readline** reads one line from the file. This method returns one line from the file and moves the file-position pointer to the next line in the file. When the file contains no more lines (i.e., the program has reached the end of the file), **readline** returns the empty string (**""**).
- In the formatted input/output model, fields—and hence records—can vary in size. Therefore, the formatted input/output model usually is not used to update records.
- Sequential-access files are inappropriate for so-called “instant-access” applications in which a particular record of information must be located immediately.
- Individual records of a random-access file can be accessed directly (and quickly) without searching through other records. Random-access files sometimes are called direct-access files.
- Data can be inserted into a random-access file without destroying other data in the file. In addition, previously stored data can be updated or deleted without rewriting the entire file.

- Module **shelve** reads and writes objects to a random-access file.
- A program can create **shelve** file objects to represent records. These objects have a dictionary interface—the record key accesses a record's information.
- The **shelve.open** function resembles the Python function **open** that opens regular files.
- Method **keys** returns a list of the record keys in the **shelve** file.
- Serialization, or pickling, converts complex object types, such as programmer-defined classes, to sets of bytes for storage or for transmission over a network. Pickling also is referred to as flattening or marshalling.
- Python provides both modules **pickle** and **Pickle** to perform pickling.
- Module **cPickle**, which is implemented in C, executes much faster than does module **pickle**, which is implemented in Python. Modules written in compiled languages, such as C, execute faster than do interpreted languages such as Python.
- Function **cPickle.dump** pickles objects.
- Unpickling restores a pickled object to its original form.
- Function **cPickle.load** unpickles pickled objects in a file.

TERMINOLOGY

>> symbol
"a" file-open mode
"ab" file-open mode
bit
buffering mode
byte
character set
close method
cPickle method
data hierarchy
database
database management system (DBMS)
end-of-file marker
EOFError exception
field
file
file name
file-open mode
file-position pointer
file-seek location
instant-access processing
IOError exception
keys method
magnetic disk
offset
open method
persistent data
pickling an object
random-access file
raw data processing
readline method
readlines method
record
record key
redirection of output
"r" file-open mode
"r+" file-open mode
"r+b" file-open mode
"rb" file-open mode
seek method
sequential-access file
serialization
shelve file
shelve module
split method
standard-error stream (**sys.stderr**)
standard-input stream (**sys.stdin**)
standard-output stream (**sys.stdout**)
stream
sys.exit function
tell method
transaction-processing systems
truncate
unpickling an object
"w" file-open mode
"w+" file-open mode
"w+b" file-open mode
"wb" file-open mode

SELF-REVIEW EXERCISES

14.1 Fill in the blanks in each of the following statements:

a) Computers store files on __________, such as magnetic disks.
b) A record can be implemented as a __________, a __________ or a __________ in Python.
c) The set of all characters used to write programs on a computer is called its __________.
d) In a __________, records typically are stored in order by the record key.
e) Python creates three file streams—__________, __________ and __________.
f) A __________ is composed of several fields.
g) To facilitate the retrieval of specific records from a file, one field in each record is chosen as a __________.
h) At the lowest level, the functions performed by computers essentially involve the manipulation of __________ and __________.
i) Data items represented in computers form a __________, in which data items become larger and more complex as they progress from bits to fields.
j) A group of related files is called a __________.

14.2 State which of the following are *true* and which are *false*. If *false*, explain why.

a) The programmer must create the **`sys.stderr`** stream explicitly.
b) The smallest data item in a computer is a byte.
c) Python views each file as a dictionary.
d) File streams serve as communication channels.
e) It is not necessary to search through all the records in a random-access file to find a specific record.
f) Records in random-access files must be of uniform length.
g) Module **`cPickle`** performs more efficiently than does module **`pickle`** because **`cPickle`** is written in Python.
h) Serialization converts complex objects to a set of bytes.
i) Method **`sys.exit`** returns **`1`** by default to signify that no errors occurred.
j) Sequential-access files are inappropriate for instant-access applications in which records must be located quickly.

ANSWERS TO SELF-REVIEW EXERCISES

14.1 a) secondary storage devices. b) tuple, dictionary, instance. c) character set. d) random-access file. e) **`sys.stdout`**, **`sys.stdin`**, **`sys.stderr`**. f) record. g) record key. h) **`0`**'s, **`1`**'s. i) data hierarchy. j) database.

14.2 a) False. This stream is created for the programmer. b) False. The smallest data item in a computer is a bit. c) False. Python views each file as a stream. d) True. e) True. f) False. Records in random-access files normally are of uniform length, but are not required to be so. g) False. Module **`cPickle`** performs more efficiently because it is written in C and compiled into native machine language for each platform. h) True. i) False. Method **`sys.exit`** returns **`0`** by default to signify that no errors occurred. j) True.

EXERCISES

14.3 Fill in the blanks in each of the following statements:

a) A group of related characters that conveys meaning is called a __________.
b) Method __________ repositions the file-position pointer in a file.
c) Programs can __________ output from the print statement to print to a different file object.
d) If the user enters the end-of-file character, function raw_input raises an __________.
e) Method __________ returns a list of the lines in a file.

14.4 State which of the following are *true* and which are *false*. If *false*, explain why.

a) People prefer to manipulate bits instead of characters and fields because bits are more compact.
b) People specify programs and data items as characters; computers then manipulate and process these characters as groups of zeros and ones.
c) Most organizations store all information in a single file to facilitate computer processing.
d) Each statement that processes a file in a Python program explicitly refers to that file by name.
e) Python imposes no structure on a file.

14.5 You are the owner of a hardware store and need to keep an inventory that can tell you what different tools you have, how many of each you have on hand and the cost of each one. Write a program that initializes the **shelve** file **"hardware.dat"**, lets you input the data concerning each tool and enables you to list all your tools. The tool identification number should be the record number. Use the following information to start your file:

14.6 Modify the inventory program of Exercise 14.5. The modified program allows you to delete a record for a tool that you no longer have and allows you to update *any* information in the file.

14.7 Create a simple text editor GUI that allows the user to open a file. The GUI should display the text of the file and then close the file. The user can modify the file's contents. When the user chooses to save the text, the modified contents should be written to the file, replacing any other contents. The user also should be able to clear the display.

14.8 Create four band members from the class **BandMember**. Pickle these objects and store them in a file. Unpickle, then output the objects.

Record Number	Tool name	Quantity	Cost
17	Hammer	76	11.99
37	Saw	88	12.00
68	Screwdriver	106	6.99
83	Wrench	34	7.50

Fig. 14.14 Data for Exercise 14.5.

```
class BandMember:
   """Represent a band member"""

   def __init__( self, name, instrument ):
      """Initialize name and instrument"""

      self.name = name
      self.instrument = instrument

   def __str__( self ):
      """Overloaded string representation"""

      return "%s plays the %s" % ( self.name, self.instrument )
```

Fig. 14.15 Class **BandMember**.

15

Extensible Markup Language (XML)

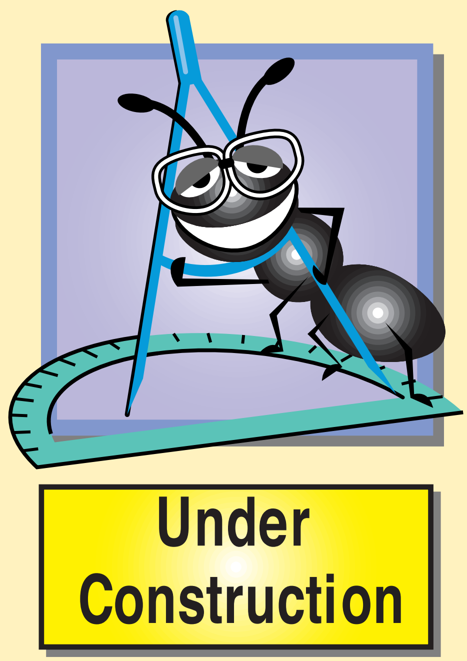
Objectives

- To understand XML.
- To mark up data using XML.
- To become familiar with the types of markup languages created with XML.
- To understand the relationships among DTDs, Schemas and XML.
- To understand the fundamentals of DOM-based and SAX-based parsing.
- To understand the concept of XML namespaces.
- To create simple XSLT documents.

Every country has its own language, yet the subjects of which the untutored soul speaks are the same everywhere.
Tertullian

The chief merit of language is clearness, and we know that nothing detracts so much from this as do unfamiliar terms.
Galen

Like everything metaphysical, the harmony between thought and reality is to be found in the grammar of the language.
Ludwig Wittgenstein



Outline

15.1 Introduction

The *Extensible Markup Language* (XML) was developed in 1996 by the *World Wide Web Consortium's (W3C's) XML Working Group*. XML is a portable, widely supported, *open technology* (i.e., non-proprietary technology) for describing data. XML quickly is becoming the standard for data that is exchanged between applications. Using XML, document authors can describe any type of data, including mathematical formulas, software configuration instructions, music, recipes and financial reports. An additional benefit of using XML is that documents are readable by both humans and machines.

This chapter explores XML and various XML-related technologies. The first three sections introduce XML and how it is used to mark up data. The next two sections describe two different programmatic libraries that can be used to manipulate XML documents. Later sections introduce several *XML vocabularies* (i.e., markup languages created with XML). This chapter also examines a technology called Extensible Stylesheet Language Transformations (XSLT), which transforms XML data into other text-based formats. Chapter 16, Python XML Processing, builds upon the concepts presented in this chapter by writing Python applications that use XML.

15.2 XML Documents

Our first XML document describes an article (Fig. 15.1). [*Note*: Every XML document we show has line numbers for the reader's convenience. These line numbers are not part of the XML documents.]

This document begins with an optional *XML declaration* (line 1), which identifies the document as an XML document. The `version` *information parameter* specifies the version

```
1  <?xml version = "1.0"?>
2  
3  <!-- Fig. 15.1: article.xml       -->
4  <!-- Article structured with XML. -->
5  
6  <article>
7  
8     <title>Simple XML</title>
9  
10    <date>December 21, 2001</date>
11 
12    <author>
13       <firstName>John</firstName>
14       <lastName>Doe</lastName>
15    </author>
16 
17    <summary>XML is pretty easy.</summary>
18 
19    <content>In this chapter, we present a wide variety of examples
20       that use XML.
21    </content>
22 
23 </article>
```

Fig. 15.1 XML used to mark up an article.

of XML[1] that is used in the document. XML comments (lines 3–4) begin with `<!--` and end with `-->`, and can be placed almost anywhere in an XML document. As in a Python program, comments are used in XML for documentation purposes.

Common Programming Error 15.1

Placing any characters, including whitespace, before the XML declaration is an error.

Portability Tip 15.1

Although the XML declaration is optional, documents should include the declaration to identify the version of XML used. Otherwise, in the future, a document that lacks an XML declaration might be assumed to conform to the latest version of XML, and errors could result.

XML marks up data using *tags*, which are names enclosed in *angle brackets* (`<>`). Tags are used in pairs to delimit character data (e.g., **`Simple XML`**). A tag that begins *markup* (i.e., XML data) is called a *start tag*, whereas a tag that terminates markup is called an *end tag*. Examples of start tags are **`<article>`** and **`<title>`** (lines 6 and 8, respectively). End tags differ from start tags in that they contain a *forward slash* (`/`) character immediately after the `<` character. Examples of end tags are **`</title>`** and **`</article>`** (lines 8 and 23, respectively). XML documents can contain any number of tags.

Common Programming Error 15.2

Failure to provide a corresponding end tag for a start tag is an error.

1. Currently, there is only one version of XML, 1.0.

Individual units of markup (i.e., everything included between a start tag and its corresponding end tag) are called *elements*. An XML document includes one element (called a *root element*) that contains all other elements in the document. The root element must be the first element after the XML declaration. In Fig. 15.1, **`article`** (line 6) is the root element. Elements are *nested* to form hierarchies—with the root element at the top of the hierarchy. This allows document authors to create explicit relationships between data. For example, elements **`title`**, **`date`**, **`author`**, **`summary`** and **`content`** then are nested within **`article`**. Elements **`firstName`** and **`lastName`** are nested within **`author`**.

Common Programming Error 15.3

Attempting to create more than one root element in an XML document is an error.

Element **`title`** (line 8) contains the title of the article, **`Simple XML`**, as character data. Similarly, **`date`** (line 10), **`summary`** (line 17) and **`content`** (lines 19–21) contain character data that represent the article's publication date, summary and content, respectively. XML tag names can be of any length and may contain letters, digits, underscores, hyphens and periods—they must begin with a letter or an underscore.

Common Programming Error 15.4

XML is case sensitive. Using the wrong case for an XML tag name is an error.

By itself, this document is simply a text file named **`article.xml`**. Although it is not required, most XML-document file names end with the file extension ***`.xml`***.[2] Processing an XML document requires a program called an *XML parser*. Parsers are responsible for checking an XML document's syntax and making the XML document's data available to applications. Often, XML parsers are built into applications or available for download over the Internet. Popular parsers include Microsoft's *msxml*, ***`4DOM`*** (a Python package that we use extensively in the Chapter 16), the Apache Software Foundation's *Xerces* and IBM's *XML4J*. In this chapter, we use msxml.

When the user loads **`article.xml`** into Internet Explorer (IE),[3] msxml parses the document and passes the parsed data to IE. IE then uses a built-in *style sheet* to format the data. Notice that the resulting format of the data (Fig. 15.2) is similar to the format of the XML document shown in Fig. 15.1. As we soon demonstrate, style sheets play an important and powerful role in the transformation of XML data into formats suitable for display.

Notice the minus (**–**) and plus (**+**) signs in Fig. 15.2. Although these are not part of the XML document, IE places them next to all *container elements* (i.e., elements that contain other elements). Container elements also are called *parent elements*. A minus sign indicates that the parent element's *child elements* (i.e., nested elements) currently are displayed. When clicked, a minus sign becomes a plus sign (which collapses the container element and hides all of its children). Conversely, clicking a plus sign expands the container element and changes the plus sign to a minus sign. This behavior is similar to the viewing of the directory structure on a Windows system using Windows Explorer. In fact, a directory structure often is modeled as a series of tree structures, in which each drive letter (e.g., **`C:`**, etc.) represents

2. Some applications that process XML documents may require this file extension.
3. IE 5 and higher.

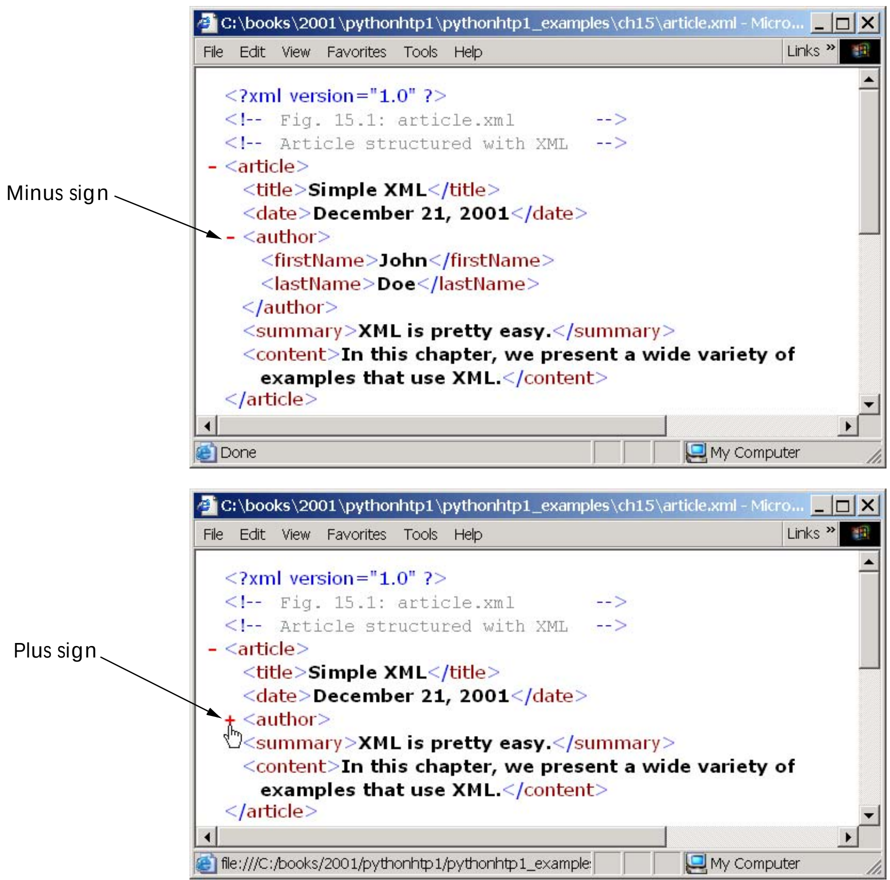


Fig. 15.2 `article.xml` displayed by Internet Explorer.

the *root* of a tree. Each folder is a *node* in the tree. Parsers often place XML data into trees to facilitate efficient manipulation, as discussed in Section 15.4.

Common Programming Error 15.5

Nesting XML tags improperly is an error. For example, **`<x><y>hello</x></y>`** *is an error, because the* **`</y>`** *tag must precede the* **`</x>`** *tag.*

We now present a second XML document (Fig. 15.3), which marks up a business letter. This document contains significantly more data than did the previous XML document.

```
<?xml version = "1.0"?>

<!-- Fig. 15.3: letter.xml                -->
<!-- Business letter formatted with XML. -->

```

Fig. 15.3 Business letter marked up as XML. (Part 1 of 2.)

```
6  <letter>
7     <contact type = "from">
8        <name>Jane Doe</name>
9        <address1>Box 12345</address1>
10       <address2>15 Any Ave.</address2>
11       <city>Othertown</city>
12       <state>Otherstate</state>
13       <zip>67890</zip>
14       <phone>555-4321</phone>
15       <flag gender = "F" />
16    </contact>
17
18    <contact type = "to">
19       <name>John Doe</name>
20       <address1>123 Main St.</address1>
21       <address2></address2>
22       <city>Anytown</city>
23       <state>Anystate</state>
24       <zip>12345</zip>
25       <phone>555-1234</phone>
26       <flag gender = "M" />
27    </contact>
28
29    <salutation>Dear Sir:</salutation>
30
31       <paragraph>It is our privilege to inform you about our new
32       database managed with <technology>XML</technology>. This
33       new system allows you to reduce the load on
34       your inventory list server by having the client machine
35       perform the work of sorting and filtering the data.
36       </paragraph>
37
38       <paragraph>Please visit our Web site for availability
39       and pricing.
40       </paragraph>
41
42    <closing>Sincerely</closing>
43
44    <signature>Ms. Doe</signature>
45 </letter>
```

Fig. 15.3 Business letter marked up as XML. (Part 2 of 2.)

Root element **`letter`** (lines 6–45) contains the child elements **`contact`** (lines 7–16 and 18–27), **`salutation`**, **`paragraph`**, **`closing`** and **`signature`**. In addition to being placed between tags, data also can be placed in *attributes*, which are name-value pairs in start tags. Elements can have any number of attributes in their start tags. The first **`contact`** element (lines 7–16) has attribute **`type`** with attribute *value* **`"from"`**, which indicates that this **`contact`** element marks up information about the letter's sender. The second **`contact`** element (lines 18–27) has attribute **`type`** with value **`"to"`**, which indicates that this **`contact`** element marks up information about the letter's recipient. Like tag names, attribute names are case sensitive; can be any length; may contain letters, digits, underscores, hyphens and periods; and must begin with either a letter or underscore char-

acter. A **contact** element stores a contact's name, address and phone number. Element **salutation** (line 29) marks up the letter's salutation. Lines 31–40 mark up the letter's body with **paragraph** elements. Elements **closing** (line 42) and **signature** (line 44) mark up the closing sentence and the signature of the letter's author, respectively.

Common Programming Error 15.6

*Failure to enclose attribute values in double (***""***) or single (***''***) quotes is an error.*

In line 15, we introduce *empty element* **flag**, which indicates the gender of the contact. Empty elements do not contain character data (i.e., they do not contain text between the start and end tags). Such elements are closed either by placing a slash at the end of the element (as shown in line 15) or by explicitly writing a closing tag, as in

```
<flag gender = "F"></flag>
```

15.3 XML Namespaces

Languages such as Python provide massive class libraries that group their features into namespaces. These namespaces prevent *naming collisions* between programmer-defined identifiers and identifiers in class libraries. For example, we might use class **Book** to represent information on one of our publications; however, a stamp collector might use class **Book** to represent a book of stamps. Without using namespaces to differentiate the two **Book** classes, a naming collision would occur if we use these two classes in the same application.

Like Python, XML also provides *namespaces* for unique identification of XML elements. In addition, XML-based languages—called *vocabularies*, such as XML Schema (Section 15.6) and the Extensible Stylesheet Language (Section 15.8)—often use namespaces to identify their elements.

Namespace prefixes, which identify the namespace to which an element belongs, differentiate elements. For example,

```
<deitel:publication>
   Python How to Program
</deitel:publication>
```

qualifies element **publication** with namespace prefix **deitel**. This indicates that element **publication** is part of namespace **deitel**. Document authors can use any name for a namespace prefix except the reserved namespace prefix ***xml***.

Common Programming Error 15.7

Attempting to create a namespace prefix named **xml** *in any combination of uppercase and lowercase letters is an error.*

The markup in Fig. 15.4 demonstrates the use of namespaces. This XML document contains two **file** elements that are differentiated using namespaces.

```
<?xml version = "1.0"?>

<!-- Fig. 15.4: namespace.xml  -->
<!-- Demonstrating namespaces. -->
```

Fig. 15.4 XML namespaces demonstration. (Part 1 of 2.)

```
5
6   <text:directory xmlns:text = "http://www.deitel.com/ns/python1e"
7      xmlns:image = "http://www.deitel.com/images/ns/120101">
8
9      <text:file filename = "book.xml">
10        <text:description>A book list</text:description>
11     </text:file>
12
13     <image:file filename = "funny.jpg">
14        <image:description>A funny picture</image:description>
15        <image:size width = "200" height = "100" />
16     </image:file>
17
18  </text:directory>
```

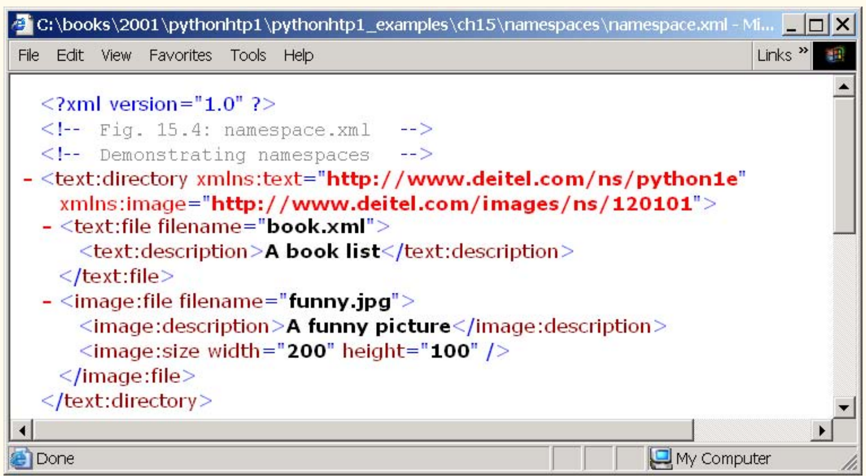


Fig. 15.4 XML namespaces demonstration. (Part 2 of 2.)

Software Engineering Observation 15.1

Attributes need not be qualified with namespace prefixes, because they always are associated with elements.

Lines 6–7 use attribute **`xmlns`** to create two namespace prefixes: **`text`** and **`image`**. Each namespace prefix is bound to a series of characters called a *uniform resource identifier (URI)* that uniquely identifies the namespace. Document authors create their own namespace prefixes and URIs.

To ensure that namespaces are unique, document authors must provide unique URIs. Here, we use the text **`http://www.deitel.com/ns/python1e`** and **`http://www.deitel.com/images/ns/120101`** as URIs. A common practice is to use *Universal Resource Locators (URLs)* for URIs, because the domain names (such as, **`www.deitel.com`**) used in URLs are guaranteed to be unique. In this example, we use URLs related to the Deitel & Associates, Inc., domain name to identify namespaces. The parser never visits these URLs—they simply represent a series of characters used to differentiate names. The URLs need not refer to actual Web pages or be formed properly.

Lines 9–11 use the namespace prefix **`text`** to describe elements **`file`** and **`description`**. Notice that the namespace prefix **`text`** is applied to the end tag name as well. Lines 13–16 apply namespace prefix **`image`** to elements **`file`**, **`description`** and **`size`**.

To eliminate the need to precede each tag name with a namespace prefix, document authors can specify a *default namespace*. Figure 15.5 demonstrates the creation and use of default namespaces.

Line 6 defines a default namespace by binding a URI to attribute **`xmlns`**. Once this default namespace is defined, tag names in child elements belonging to the namespace need not be qualified by a namespace prefix. Element **`file`** (line 9–11) is in the namespace corresponding to the URI **`http://www.deitel.com/ns/python1e`**. Compare this to lines 9–11 of Fig. 15.4, where we prefixed elements **`file`** and **`description`** with **`text`**.

The default namespace applies to element **`directory`** and all elements that are not qualified with a namespace prefix. However, we can use a namespace prefix to specify a different namespace for particular elements. For example, line 13 prefixes tag name **`file`**

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.5: defaultnamespace.xml -->
4  <!-- Using default namespaces.       -->
5
6  <directory xmlns = "http://www.deitel.com/ns/python1e"
7     xmlns:image = "http://www.deitel.com/images/ns/120101">
8
9     <file filename = "book.xml">
10       <description>A book list</description>
11    </file>
12
13    <image:file filename = "funny.jpg">
14       <image:description>A funny picture</image:description>
15       <image:size width = "200" height = "100" />
16    </image:file>
17
18 </directory>
```

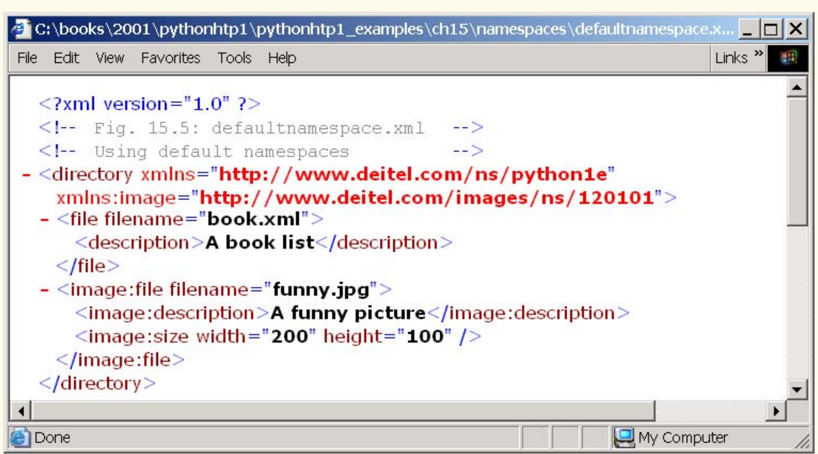


Fig. 15.5 Default namespace demonstration.

in with **`image`** to indicate that it is in the namespace corresponding to the URI **`http://www.deitel.com/images/ns/120101`**, rather than in the default namespace.

15.4 Document Object Model (DOM)

Although XML documents are text files, retrieving data from them via sequential-file access techniques is neither practical nor efficient, especially in situations where data must be added or deleted dynamically.

Upon successful parsing of documents, some XML parsers store document data as tree structures in memory. Figure 15.6 illustrates the tree structure for the document **`article.xml`** discussed in Fig. 15.1. This hierarchical tree structure is called a *Document Object Model (DOM)* tree, and an XML parser that creates this type of structure is known as a *DOM parser*. The DOM tree represents each component of the XML document (e.g., **`article`**, **`date`**, **`firstName`**, etc.) as a node in the tree. Nodes (such as, **`author`**) that contain other nodes (called *child nodes*) are called *parent nodes*. Nodes that have the same parent (such as, **`firstName`** and **`lastName`**) are called *sibling nodes*. A node's *descendant nodes* include that node's children, its children's children and so on. Similarly, a node's *ancestor nodes* include that node's parent, its parent's parent and so on.

The DOM has a single *root node*, called the *document root*, which contains all other nodes in a document. For example, the root node for **`article.xml`** (Fig. 15.1) contains a node for the XML declaration (line 1), two nodes for the comments (lines 3–4) and a node for the root element (line 6).

Each node is an object that has attributes and methods. Attributes associated with a node include tag names, values, child nodes, etc. Methods enable programs to create, delete and append nodes, load XML documents and so on. The XML parser exposes these methods as a programmatic library—called an *Application Programming Interface* (*API*). We discuss how to use the DOM API in Chapter 16, Python XML Processing.

15.5 Simple API for XML (SAX)

Members of the *XML-DEV mailing list* developed the Simple API for XML (SAX), which they released in May, 1998. SAX is an alternate method for parsing XML documents that

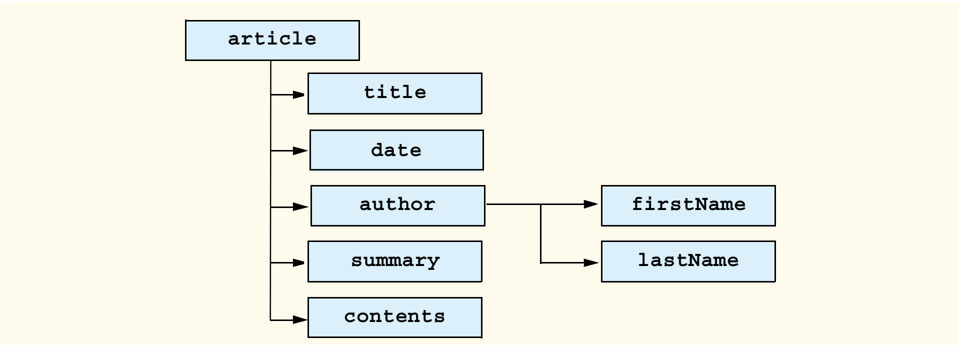


Fig. 15.6 Tree structure for **`article.xml`**.

uses an *event-based model*—*SAX-based parsers* generate notifications called *events* as the parser processes the document. Software programs can “listen” for these events to retrieve data from the document. For example, a program that builds mailing lists might read name and address information from an XML document that contains much more than just mailing address information (e.g., birthdays, phone numbers, email addresses, etc.). Such a program could use a SAX parser to parse the document, and might listen only for events that contain name and address information. SAX-based parsers are available for a variety of programming languages such as Python, Java and C++. We demonstrate SAX-based parsing in Chapter 16, Python XML Processing.

SAX and DOM provide dramatically different APIs for accessing XML document data. Each API has advantages and disadvantages. DOM is a tree-based model that stores the document’s data in a hierarchy of nodes. Programs can access data quickly, because all the document’s data is in memory. DOM also provides facilities for adding or removing nodes, which enables programs to modify XML documents easily.

SAX-based parsers invoke *listener methods* when the parser encounters markup. With this event-based model, the SAX-based parser does not create a tree structure to store the XML document’s data—instead, the parser passes data to the application from the XML document as the parser finds that data. This results in greater performance and less memory overhead than with DOM-based parsers. In fact, many DOM parsers use SAX parsers “under the hood” to retrieve data from a document for building the DOM tree in memory. Many programmers find it easier to traverse and manipulate XML documents using the DOM tree structure. As a result, programs typically use SAX parsers for reading XML documents that the program will not modify.

Performance Tip 15.1

SAX-based parsing often is more efficient than DOM-based parsing for processing large XML documents—SAX-based parsers do not load entire XML documents into memory.

Performance Tip 15.2

SAX-based parsing is an efficient means of parsing documents that only need parsing once.

Performance Tip 15.3

DOM-based parsing often is more efficient than SAX-based parsing when a program must retrieve specific information from the document quickly.

Performance Tip 15.4

Programs that must conserve memory commonly use SAX-based parsers.

Software Engineering Observation 15.2

Members of the XML-DEV mailing list developed SAX independently of the W3C, although SAX has wide industry support. DOM is the official W3C recommendation.

15.6 Document Type Definitions (DTDs), Schemas and Validation

This section introduces *Document Type Definitions* (*DTDs)* and *Schemas*—documents that specify the structure of XML documents (i.e., what elements are permitted, what attributes an element can have and so on). When a DTD or Schema document is provided, some parsers

(called *validating parsers*[4]) read the DTD or Schema and check the XML document's structure against it. If the XML document conforms to the DTD or Schema, then the XML document is *valid*. Parsers that cannot validate documents against DTDs or Schemas are called *non-validating parsers*. If an XML parser (validating or non-validating) is able to process an XML document (that does not reference a DTD or Schema), the XML document is considered to be *well formed* (i.e., it is syntactically correct). By definition, a valid XML document is a well-formed XML document. If a document is not well formed, the parser issues an error.

Software Engineering Observation 15.3

DTD and Schema documents are essential components for XML documents used in business-to-business (B2B) transactions and mission-critical systems.

Software Engineering Observation 15.4

Because XML document content can be structured in many different ways, an application cannot determine whether the document data it receives is complete, missing data or ordered properly. DTDs and Schemas solve this problem by providing an extensible means of describing a document's contents. An application can use a DTD or Schema document to perform a validity check on the document's contents.

15.6.1 Document Type Definition Documents

Document type definitions (DTDs) provide a means for type checking XML documents and thus verifying their *validity* (confirming that elements contain the proper attributes, elements are in the proper sequence, etc.). DTDs use *EBNF* (*Extended Backus-Naur Form*) *grammar* to describe an XML document's content. XML parsers need additional functionality to read EBNF grammar, because it is not XML syntax. Although DTDs are optional, they are recommended to ensure document conformity. The DTD in Fig. 15.7 defines the set of rules (i.e., the grammar) for structuring the business letter document contained in Fig. 15.8.

```
<!-- Fig. 15.7: letter.dtd       -->
<!-- DTD document for letter.xml. -->

<!ELEMENT letter ( contact+, salutation, paragraph+,
   closing, signature )>

<!ELEMENT contact ( name, address1, address2, city, state,
   zip, phone, flag )>
<!ATTLIST contact type CDATA #IMPLIED>

<!ELEMENT name ( #PCDATA )>
<!ELEMENT address1 ( #PCDATA )>
<!ELEMENT address2 ( #PCDATA )>
<!ELEMENT city ( #PCDATA )>
<!ELEMENT state ( #PCDATA )>
<!ELEMENT zip ( #PCDATA )>
<!ELEMENT phone ( #PCDATA )>
```

Fig. 15.7 Document Type Definition (DTD) for a business letter. (Part 1 of 2.)

4. Many DOM parsers and SAX parsers are validating parsers. Check you parser's documentation to determine whether it is a validating parser.

```
18   <!ELEMENT flag EMPTY>
19   <!ATTLIST flag gender (M | F) "M">
20
21   <!ELEMENT salutation ( #PCDATA )>
22   <!ELEMENT closing ( #PCDATA )>
23   <!ELEMENT paragraph ( #PCDATA )>
24   <!ELEMENT signature ( #PCDATA )>
```

Fig. 15.7 Document Type Definition (DTD) for a business letter. (Part 2 of 2.)

Portability Tip 15.2

DTDs can ensure consistency among XML documents generated by different programs.

Line 4 uses the ***ELEMENT*** *element type declaration* to define rules for element **letter**. In this case, **letter** contains one or more **contact** elements, one **salutation** element, one or more **paragraph** elements, one **closing** element and one **signature** element, in that sequence. The *plus sign* (**+**) *occurrence indicator* specifies that an element must occur one or more times. Other indicators include the *asterisk* (*****), which indicates an optional element that can occur any number of times, and the *question mark* (**?**), which indicates an optional element that can occur at most once. If an occurrence indicator is omitted, exactly one occurrence is expected.

The **contact** element definition (line 7) specifies that it contains the **name**, **address1**, **address2**, **city**, **state**, **zip**, **phone** and **flag** elements—in that order. Exactly one occurrence of each is expected.

Line 9 uses the ***ATTLIST*** *element type declaration* to define an attribute (i.e., **type**) for the **contact** element. Keyword ***#IMPLIED*** specifies that, if the parser finds a **contact** element without a **type** attribute, the application can provide a value or ignore the missing attribute. The absence of a **type** attribute cannot invalidate the document. Other types of default values include ***#REQUIRED*** and ***#FIXED***. Keyword **#REQUIRED** specifies that the attribute must be present in the document and the keyword **#FIXED** specifies that the attribute (if present) must always be assigned a specific value. For example,

```
<!ATTLIST address zip #FIXED "01757">
```

indicates that the value **01757** must be used for attribute **zip**; otherwise, the document is invalid. If the attribute is not present, then the parser, by default, uses the fixed value that is specified in the **ATTLIST** declaration. Flag ***CDATA*** specifies that attribute **type** contains text that is not processed by the parser, but instead is passed to the application as is.

Software Engineering Observation 15.5

DTD syntax cannot describe an element's (or attribute's) data type.

Flag ***#PCDATA*** (line 11) specifies that the element can store *parsed character data* (i.e., text). Parsed character data cannot contain markup. Because they are used in markup, the characters less than (**<**) and ampersand (**&**) must be replaced by their *entity references* (i.e., **<** and **&**). However, the ampersand character can be used with entity references. See Appendix M, HTML/XHTML Special Characters, for a list of pre-defined entities.

Line 18 defines an empty element named **flag**. Keyword ***EMPTY*** specifies that the element cannot contain character data. Empty elements commonly are used for their attributes.



Common Programming Error 15.8

Any element, attribute or relationship not explicitly defined by a DTD results in an invalid document.

XML documents must reference a DTD explicitly. Figure 15.8 is an XML document that conforms to **letter.dtd** (Fig. 15.7).

```
<?xml version = "1.0"?>

<!-- Fig. 15.8: letter2.xml                -->
<!-- Business letter formatted with XML. -->

<!DOCTYPE letter SYSTEM "letter.dtd">

<letter>
   <contact type = "from">
      <name>Jane Doe</name>
      <address1>Box 12345</address1>
      <address2>15 Any Ave.</address2>
      <city>Othertown</city>
      <state>Otherstate</state>
      <zip>67890</zip>
      <phone>555-4321</phone>
      <flag gender = "F" />
   </contact>

   <contact type = "to">
      <name>John Doe</name>
      <address1>123 Main St.</address1>
      <address2></address2>
      <city>Anytown</city>
      <state>Anystate</state>
      <zip>12345</zip>
      <phone>555-1234</phone>
      <flag gender = "M" />
   </contact>

   <salutation>Dear Sir:</salutation>

      <paragraph>It is our privilege to inform you about our new
      database managed with XML. This new system
      allows you to reduce the load on your inventory list
      server by having the client machine perform the work of
      sorting and filtering the data.
      </paragraph>

      <paragraph>Please visit our Web site for availability
      and pricing.
      </paragraph>
   <closing>Sincerely</closing>
   <signature>Ms. Doe</signature>
</letter>
```

Fig. 15.8 XML document referencing its associated DTD.

This XML document is similar to that in Fig. 15.3. Line 6 references a DTD file. This markup contains three pieces: The name of the root element (**`letter`** in line 8) to which the DTD is applied, the keyword ***`SYSTEM`*** (which in this case denotes an *external DTD*—a DTD defined in a separate file) and the DTD's name and location (i.e., **`letter.dtd`** in the current directory). Though almost any file extension can be used, DTD documents typically end with the ***`.dtd`*** extension.

Various tools (many of which are free) check document conformity against DTDs and Schemas (discussed momentarily). The output in Fig. 15.9 shows the results of validating **`letter2.xml`** against **`letter.dtd`** using Microsoft's *XML Validator*. Microsoft XML Validator is available free for download from **`msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp`**. For additional validation tools, visit **`www.w3.org/XML/Schema.html`**.

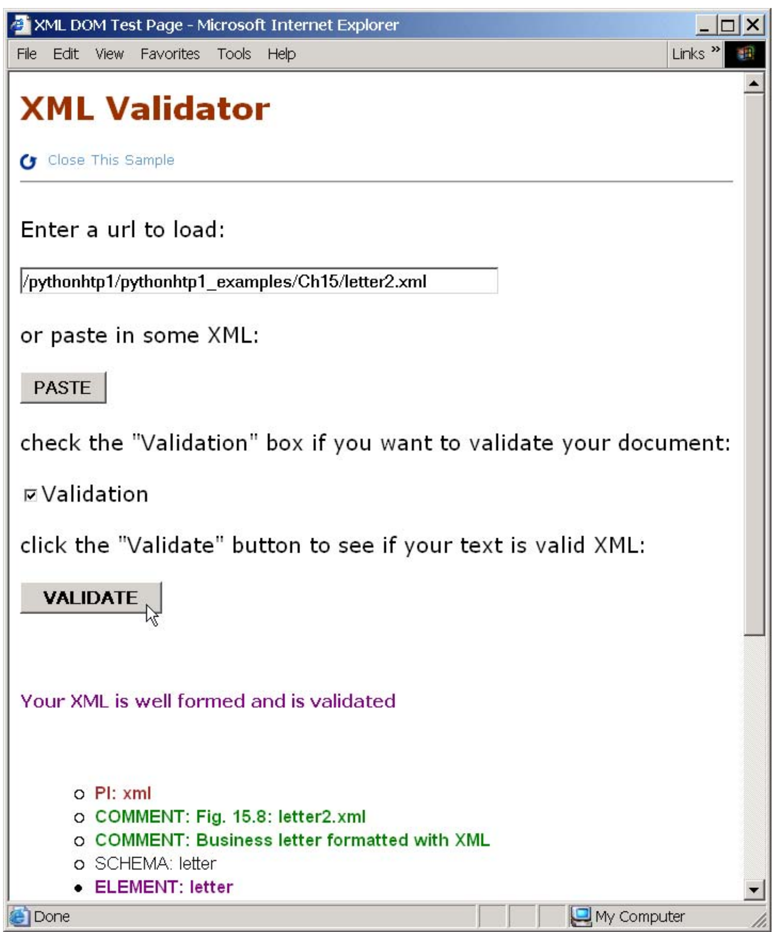


Fig. 15.9 XML Validator validating an XML document against a DTD.

The Microsoft XML Validator can validate XML documents against DTDs locally or by uploading the documents to the XML Validator Web site. Here, **letter2.xml** and **letter.dtd** are placed in folder **/pythonhtp1/pythonhtp1_examples/Ch15**. This XML document (**letter2.xml**) is valid because it conforms to **letter.dtd**.

XML documents that fail validation still may be well-formed documents. When a document fails to conform to a DTD or Schema, Microsoft XML Validator displays an error message. For example, the DTD in Fig. 15.8 indicates that the **contacts** element must contain child element **name**. If this element is omitted, the document is well formed, but not valid. In such a scenario, Microsoft XML Validator displays the error message shown in Fig. 15.10.

15.6.2 W3C XML Schema Documents

This section introduces *W3C XML Schema*[5]—a W3C *Recommendation* (i.e., a stable release suitable for use in industry). Many developers in the XML community believe DTDs are not flexible enough to meet today's programming needs. For example, programs cannot manipulate DTDs (e.g., search, transform into different representations such as XHTML, etc.) in the same manner as XML documents because DTDs are not themselves XML documents. These and other limitations led to the development of Schemas.

Unlike DTDs, Schemas do not use EBNF grammar. Instead, Schemas use XML syntax and are actually XML documents that can be manipulated programmatically. Like DTDs, Schemas require validating parsers. In the near future, Schemas likely will replace DTDs as the primary means of describing XML document structure.

A DTD describes an XML document's structure, not the content of that document's elements. For example,

```
<quantity>5</quantity>
```

contains character data. If the document containing element **quantity** references a DTD, an XML parser can validate the document to confirm that this element indeed does contain **PCDATA** content, but the parser cannot validate whether the content is numeric; DTDs do not provide such capability. So, unfortunately, the parser also considers markup such as

```
<quantity>hello</quantity>
```

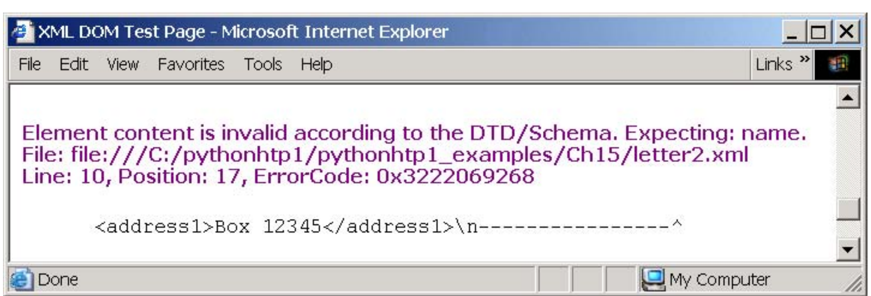


Fig. 15.10 XML Validator displaying an error message.

5. For the latest information on W3C XML Schema, visit **www.w3.org/XML/Schema**.

to be valid. The application that uses the XML document containing this markup would need to test whether the data in element **quantity** is numeric and take appropriate action if the data is not numerics.

XML Schema enables Schema authors to specify that element **quantity**'s data must be numeric. When a parser validates the XML document against this Schema, the parser can determine that **5** conforms and that **hello** does not. An XML document that conforms to a schema document is *schema valid* and a document that does not conform is invalid.

In this section, we use *XSV* (*XML Schema Validator*) to validate XML documents against W3C XML Schema. To use XSV online, visit **www.w3.org/2000/09/webdata/xsv**, enter the name of the XML file to validate, then press the **Upload and Get Results** button. Visit **www.ltg.ed.ac.uk/~ht/xsv-status.html** to download XSV.

Software Engineering Observation 15.6

Many organizations and individuals are creating DTDs and schemas for a broad range of applications (e.g., financial transactions, medical prescriptions, etc.). These collections—called repositories*—often are available free for download from the Web (e.g.,* ***www.dtd.com****).*

Figure 15.11 shows a Schema-valid XML document (**book.xml**) and Fig. 15.12 shows the W3C XML Schema document (**book.xsd**) that defines the structure for **book.xml**. W3C XML Schemas typically use the ***.xsd*** extension, although this is not required. Figure 15.11 shows the result of validating **book.xml** against Schema **book.xsd**. Note that the output is XML, and the **outcome='success'** and **schemaErrors='0'** attributes indicate that **book.xml** is valid.

W3C XML Schema use the namespace URI ***http://www.w3.org/2001/XMLSchema*** and often use *namespace prefix* ***xsd*** (line 6 in Fig. 15.12). Root element ***schema*** contains elements that define an XML document's structure. Line 7 binds the URI **http://www.deitel.com/booklist** to namespace prefix **deitel**. Line 8 specifies the ***targetNamespace***, which is the namespace for elements and attributes that this Schema defines.

Good Programming Practice 15.1

By convention, W3C XML Schema authors use namespace prefix ***xsd*** *when referring to the URI* ***http://www.w3.org/2001/XMLSchema****.*

```
<?xml version = "1.0"?>

<!-- Fig. 15.11: book.xml                         -->
<!-- Document that conforms to a W3C XML Schema. -->

<deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
   <book>
      <title>e-Business and e-Commerce How to Program</title>
   </book>
   <book>
      <title>Python How to Program</title>
   </book>
</deitel:books>
```

Fig. 15.11 XML document that conforms to a W3C XML Schema. (Part 1 of 2.)

```
C:\Program Files\XSV>xsv /pythonhtp1_examples/ch15/Schema/book.xml /
pythonhtp1_examples/ch15/Schema/book.xsd

<?xml version='1.0'?>
<xsv docElt='{http://www.deitel.com/booklist}books'
instanceAssessed='true' instanceErrors='0' rootType='{http://www.dei-
tel.com/booklist}:BooksType' schemaDocs='/pythonhtp1_examples/ch15/
Schema/book.xsd' schemaErrors='0' target='file:/pythonhtp1_examples/
ch15/Schema/book.xml' validation='strict' version='XSV 1.203.2.37/
1.106.2.19 of 2001/11/29 11:00:00'xmlns='http://www.w3.org/2000/05/
xsv'>
<schemaDocAttempt URI='file:/pythonhtp1_examples/ch15/Schema/
book.xsd' outcome='success' source='command line'/>
</xsv>
```

Fig. 15.11 XML document that conforms to a W3C XML Schema. (Part 2 of 2.)

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.12: book.xsd            -->
4  <!-- Simple W3C XML Schema document. -->
5
6  <xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
7     xmlns:deitel = "http://www.deitel.com/booklist"
8     targetNamespace = "http://www.deitel.com/booklist">
9
10    <xsd:element name = "books" type = "deitel:BooksType" />
11
12    <xsd:complexType name = "BooksType">
13       <xsd:sequence>
14          <xsd:element name = "book" type = "deitel:BookType"
15             minOccurs = "1" maxOccurs = "unbounded" />
16       </xsd:sequence>
17    </xsd:complexType>
18
19    <xsd:complexType name = "BookType">
20       <xsd:sequence>
21          <xsd:element name = "title" type = "xsd:string" />
22       </xsd:sequence>
23    </xsd:complexType>
24
25 </xsd:schema>
```

Fig. 15.12 XSD Schema document to which **book.xml** conforms.

In W3C XML Schema, element ***element*** (line 10) defines an element. Attributes ***name*** and ***type*** specify the **element**'s name and data type, respectively. In this case, the name of the element is **books** and the data type is **deitel:BooksType**. Any element (e.g., **books**) that contains attributes or child elements must define a *complex type*, which defines each attribute and child element. Type **deitel:BooksType** (lines 12–17) is an example of a complex type. We prefix **BooksType** with **deitel**, because this is a complex type that we have created, not an existing W3C XML Schema data type.

Lines 12–17 use element ***complexType*** to define an element type that has a child element named **book**. Because **book** contains a child element, its type must be a complex type (e.g., **BookType**). Attribute ***minOccurs*** specifies that **books** must contain a minimum of one **book** element. Attribute ***maxOccurs***, with value ***unbounded*** (line 14) specifies that **books** may have any number of **book** child elements. Element ***sequence*** specifies the order of elements in the complex type.

Lines 19–23 define the **complexType BookType**. Line 21 defines element **title** with **type *xsd:string***. When an element has a *simple type* such as **xsd:string**, it is prohibited from containing attributes and child elements. W3C XML Schema provides a large number of data types such as ***xsd:date*** for dates, ***xsd:int*** for integers, ***xsd:double*** for floating-point numbers and ***xsd:time*** for time.

The Schema in Fig. 15.12 indicates that every **book** element must contain child element **title**. If this element is omitted, the document is well formed, but not valid. If we remove line 8 from Fig. 15.11, XSV displays the error message shown in Fig. 15.13.

15.7 XML Vocabularies

XML allows document authors to create their own tags to describe data precisely. People and organizations in various fields of study have created many different XML vocabularies for structuring data. Some of these vocabularies are: *MathML* (*Mathematical Markup Language*), *Scalable Vector Graphics* (*SVG*), *Wireless Markup Language* (*WML*), *Extensible Business Reporting Language* (*XBRL*), *Extensible User Interface Language* (*XUL*) and *VoiceXML*™. Two other examples of XML vocabularies are W3C XML Schema and the

```
C:\PROGRA~1\XSV>xsv /pythonhtp1/pythonhtp1_examples/Ch15/Schema/
book.xml /pythonhtp1/pythonhtp1_examples/Ch15/Schema/book.xsd

<?xml version='1.0'?>
<xsv docElt='{http://www.deitel.com/booklist}books' instanceAs-
sessed='true' instanceErrors='1' rootType='{http://www.deitel.com/
booklist}:BooksType' schemaDocs='/pythonhtp1/pythonhtp1_examples/
Ch15/Schema/book.xsd' schemaErrors='0' target='file:/pythonhtp1/
pythonhtp1_examples/Ch15/Schema/book.xml' validation='strict' ver-
sion='XSV 1.203.2.37/1.106.2.19 of 2001/11/29 11:00:00' xmlns='http://
www.w3.org/2000/05/xsv'>
<schemaDocAttempt URI='file:/pythonhtp1/pythonhtp1_examples/Ch15/
Schema/book.xsd' outcome='success' source='command line'/>
<invalid char='4' code='cvc-complex-type.1.2.4' line='8' re-
source='file:/pythonhtp1/pythonhtp1_examples/Ch15/Schema/
book.xml'>content of book is not allowed to end here (1), expecting
['{None}:title']:
<fsm>
<node id='1'>
<edge dest='2' label='{None}:title'/>
</node>
<node final='true' id='2'/>
</fsm></invalid>
</xsv>
```

Fig. 15.13 XML document that does not conform to a W3C XML Schema.

Extensible Stylesheet Language (XSL), which is introduced in Section 15.8. The following subsections describe MathML, Chemical Markup Language (CML) and other XML vocabularies.

15.7.1 MathML™

Until recently, computers typically required specialized software packages such as TeX and LaTeX to display complex mathematical expressions. This section introduces MathML, which the W3C developed for describing mathematical notations and expressions. One application that can parse and render MathML is the W3C's *Amaya*™ browser/editor, which can be downloaded at no charge from

www.w3.org/Amaya/User/BinDist.html

This Web page contains download links for the Windows 95/98/NT/2000, Linux® and Solaris™ platforms. Amaya documentation and installation notes also are available at the W3C Web site.

MathML markup describes mathematical expressions for display. Figure 15.14 uses MathML to mark up a simple expression. [*Note*: In this section, we provide sample outputs that illustrate how a MathML-enabled application might render the markup.]

```
<?xml version="1.0"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 15.14: mathml1.html -->
<!-- Simple MathML.           -->

<html xmlns = "http://www.w3.org/1999/xhtml">

   <head><title>Simple MathML Example</title></head>

   <body>

      <math xmlns = "http://www.w3.org/1998/Math/MathML">

         <mrow>
            <mn>2</mn>
            <mo>+</mo>
            <mn>3</mn>
            <mo>=</mo>
            <mn>5</mn>
         </mrow>

      </math>

   </body>
</html>
```

Fig. 15.14 Expression marked up with MathML. (Part 1 of 2.)

(2 + 3 = 5)

Fig. 15.14 Expression marked up with MathML. (Part 2 of 2.)

We embed the MathML content into an XHTML document by using a ***math*** element with the default namespace ***http://www.w3.org/1998/Math/MathML*** (line 15). The ***mrow*** *element* (line 17) is a container element for expressions that contain more than one element. In this case, the **mrow** element contains five children. The ***mn*** *element* (line 18) marks up a number. The ***mo*** *element* (line 19) marks up an operator (e.g., +). Using this markup, we define the expression 2 + 3 = 5, which a software program that supports MathML could display.

Let us now consider using MathML to mark up an algebraic equation that uses exponents and arithmetic operators (Fig. 15.15).

```
<?xml version="1.0"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 15.15: mathml2.html -->
<!-- Simple MathML.           -->

<html xmlns = "http://www.w3.org/1999/xhtml">

   <head><title>Algebraic MathML Example</title></head>

   <body>

      <math xmlns = "http://www.w3.org/1998/Math/MathML">
         <mrow>

            <mrow>
               <mn>3</mn>
               <mo>&InvisibleTimes;</mo>

               <msup>
                  <mi>x</mi>
                  <mn>2</mn>
               </msup>

            </mrow>

            <mo>+</mo>
            <mi>x</mi>
            <mo>-</mo>

            <mfrac>
               <mn>2</mn>
               <mi>x</mi>
            </mfrac>
```

Fig. 15.15 Algebraic equation marked up with MathML. (Part 1 of 2.)

```
37
38                <mo>=</mo>
39                <mn>0</mn>
40
41             </mrow>
42          </math>
43
44       </body>
45    </html>
```

$$3x^2 + x - \frac{2}{x} = 0$$

Fig. 15.15 Algebraic equation marked up with MathML. (Part 2 of 2.)

Element **mrow** behaves like parentheses, which allow the document author to group related elements properly. Line 20 uses entity reference ***⁢*** to indicate a multiplication operation without a *symbolic representation* (i.e., the multiplication symbol does not appear between the **3** and **x**). For exponentiation, line 22 uses the ***msup*** element, which represents a superscript. This **msup** element has two children—the expression to be superscripted (i.e., the base) and the superscript (i.e., the exponent). Similarly, the ***msub*** element represents a subscript. To display variables such as **x**, line 23 uses *identifier element* ***mi***.

To display a fraction, line 33 uses element ***mfrac***. Lines 34–35 specify the numerator and the denominator for the fraction. If either the numerator or the denominator contains more than one element, it must be nested in an **mrow** element.

Figure 15.16 marks up a calculus expression that contains an integral symbol and a square-root symbol.

```
<?xml version="1.0"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 15.16: mathml3.html       -->
<!-- Calculus example using MathML. -->

<html xmlns = "http://www.w3.org/1999/xhtml">

   <head><title>Calculus MathML Example</title></head>

   <body>

      <math xmlns = "http://www.w3.org/1998/Math/MathML">
         <mrow>
            <msubsup>

               <mo>&Integral;</mo>
               <mn>0</mn>
```

Fig. 15.16 Calculus expression marked up with MathML. (Part 1 of 2.)

```
21
22                <mrow>
23                   <mn>1</mn>
24                   <mo>-</mo>
25                   <mi>y</mi>
26                </mrow>
27
28             </msubsup>
29
30             <msqrt>
31                <mrow>
32
33                   <mn>4</mn>
34                   <mo>&InvisibleTimes;</mo>
35
36                   <msup>
37                      <mi>x</mi>
38                      <mn>2</mn>
39                   </msup>
40
41                   <mo>+</mo>
42                   <mi>y</mi>
43
44                </mrow>
45             </msqrt>
46
47             <mo>&delta;</mo>
48             <mi>x</mi>
49          </mrow>
50       </math>
51    </body>
52 </html>
```

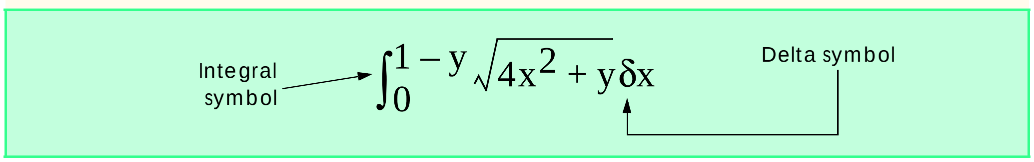


Fig. 15.16 Calculus expression marked up with MathML. (Part 2 of 2.)

The entity reference **`∫`** (line 19) represents the integral symbol, while the **`msubsup`** element (line 17) specifies the superscript and subscript. Element **`mo`** marks up the integral operator. Element **`msubsup`** requires three child elements—an operator (e.g., the integral entity reference), the subscript expression (line 20) and the superscript expression (lines 22–26). Element **`mn`** (line 20) marks up the number (i.e., **`0`**) that represents the subscript. Element **`mrow`** marks up the expression (i.e., **`1-y`**) that specifies the superscript expression

Element **`msqrt`** (lines 30–45) represents a square root expression. Line 31 uses element **`mrow`** to group the expression contained in the square root. Line 47 introduces entity reference **`δ`** for representing a delta symbol. Delta is an operator, so line 47 places this entity reference in element **`mo`**. To see other operations and symbols in MathML, visit **`www.w3.org/Math`**.

15.7.2 Chemical Markup Language (CML)

Chemical Markup Language (*CML*) is an XML vocabulary for representing molecular and chemical information. Although many of our readers will not know the chemistry required to understand the example in this section fully, we feel that CML so beautifully illustrates the purpose of XML that we chose to include the example for the readers who wish to see XML "at its best." Document authors can edit and view CML, using the *Jumbo browser*[6], which is available at **www.xml-cml.org**. Figure 15.17 shows an ammonia molecule marked up in CML.

Lines 1–2 contain a *processing instruction* (*PI*), which contains application-specific information embedded in an XML document. The characters **<?** and **?>** delimit a processing instruction. The processing instruction of lines 1–2 provides application-specific information to the Jumbo browser. Processing instructions consist of a *PI target* (e.g.,

```
1  <?jumbo:namespace ns = "http://www.xml-cml.org"
2     prefix = "C" java = "jumbo.cmlxml.*Node" ?>
3
4  <!-- Fig. 15.17: ammonia.xml -->
5  <!-- Structure of ammonia.   -->
6
7  <C:molecule id = "Ammonia">
8
9     <C:atomArray builtin = "elsym">
10       N H H H
11    </C:atomArray>
12
13    <C:atomArray builtin = "x2" type = "float">
14       1.5 0.0 1.5 3.0
15    </C:atomArray>
16
17    <C:atomArray builtin = "y2" type = "float">
18       1.5 1.5 0.0 1.5
19    </C:atomArray>
20
21    <C:bondArray builtin = "atid1">
22       1 1 1
23    </C:bondArray>
24
25    <C:bondArray builtin = "atid2">
26       2 3 4
27    </C:bondArray>
28
29    <C:bondArray builtin = "order" type = "integer">
30       1 1 1
31    </C:bondArray>
32
33 </C:molecule>
```

Fig. 15.17 CML markup for ammonia molecule. (Part 1 of 2.)

6. At the time of this writing, Jumbo did not allow users to load documents for rendering. For illustration purposes, we created the image shown in Fig. 15.17.

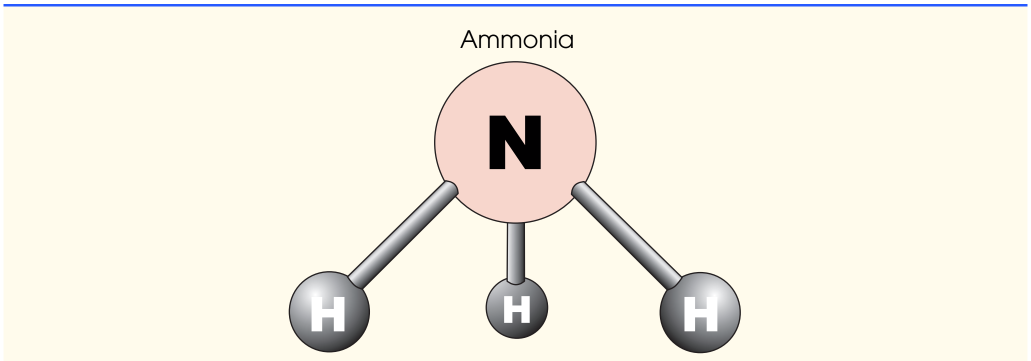


Fig. 15.17 CML markup for ammonia molecule. (Part 2 of 2.)

jumbo:namespace) and a *PI value* (e.g., **ns = "http://www.xml-cml.org" prefix = "C" java = "jumbo.cmlxml.*Node"**).

Portability Tip 15.3

Processing instructions allow document authors to embed application-specific information in an XML document, without affecting that document's portability.

Line 7 defines an ammonia molecule using element ***molecule***. Attribute ***id*** identifies this molecule as **Ammonia**. Lines 9–11 use element ***atomArray*** and attribute ***builtin*** to specify the molecule's atoms. Ammonia contains one nitrogen atom and three hydrogen atoms.

Lines 13–15 show element **atomArray** with attribute **builtin** assigned the value ***x2*** and ***type float***. This specifies that the element contains a list of floating-point numbers, each of which indicates the *x*-coordinate of an atom. The first value (**1.5**) is the *x*-coordinate of the first atom (nitrogen), the second value (**0.0**) is the *x*-coordinate of the second atom (the first hydrogen atom) and so on.

Lines 17–19 show element **atomArray** with attribute **builtin** assigned the value ***y2*** and **type float**. This specifies that the element contains a list of *y*-coordinate values. The first value (**1.5**) is the *y*-coordinate of the first atom (nitrogen), the second value (**1.5**) is the *y*-coordinate of the second atom (the first hydrogen atom) and so on.

Lines 21–23 show element ***bondArray*** with attribute **builtin** assigned the value ***atid1***. Element **bondArray** defines the bonds between atoms. This element has a **builtin** value of **atid1**, so the values this element specifies compose the first atom in a pair of atoms. We are defining three bonds, so we specify three values. For each value, we specify the first atom in the **atomArray**, the nitrogen atom.

Lines 25–27 show element **bondArray** with attribute **builtin** assigned the value ***atid2***. The values of this element compose the second atom in a pair of atoms and denote the three hydrogen atoms.

Lines 29–31 show element **bondArray** with the attribute **builtin** assigned the value ***order*** and **type *integer***. The values of this element are integers that represent the number of bonds between the pairs of atoms. Thus, the bond between the nitrogen atom and the first hydrogen is a single bond, the bond between the nitrogen atom and the second hydrogen atom is a single bond, and the bond between the nitrogen atom and the third hydrogen atom is a single bond.

15.7.3 Other XML Vocabularies

Literally hundreds of XML vocabularies derive from XML. Every day, developers find new uses for XML. In Fig. 15.18, we summarize some of these vocabularies.

15.8 Extensible Stylesheet Language (XSL)[7]

Extensible Stylesheet Language (XSL) is an XML vocabulary for formatting XML data. In this section, we discuss the portion of XSL—called *XSL Transformations* (*XSLT*)—that creates formatted text-based documents from XML documents. This process is called a *transformation* and involves two tree structures—the *source tree*, which is the XML document being transformed, and the *result tree*, which is the result (e.g., Extensible Hypertext Markup Language or XHTML[8]) of the transformation. The source tree is not modified when a transformation occurs.

Vocabulary	Description
VoiceXML™	The VoiceXML forum founded by AT&T, IBM, Lucent and Motorola developed VoiceXML. It provides interactive voice communication between humans and computers through a telephone, PDA (personal digital assistant) or desktop computer. IBM's VoiceXML SDK can process VoiceXML documents. Visit **`www.voicexml.org`** for more information on VoiceXML.
Synchronous Multimedia Integration Language (SMIL™)	SMIL is an XML vocabulary for multimedia presentations. The W3C was the primary developer of SMIL, with contributions from other companies. Visit **`www.w3.org/AudioVideo`** for more on SMIL.
Research Information Exchange Markup Language (RIXML)	RIXML, which a consortium of brokerage firms developed, marks up investment data. Visit **`www.rixml.org`** for more information on RIXML.
ComicsML	A language developed by Jason MacIntosh for marking up comics. Visit **`www.jmac.org/projects/comics_ml`** for more information on ComicsML.
Geography Markup Language (GML)	The OpenGIS developed the GML to describe geographic information. Visit **`www.opengis.org`** for more information on GML.
Extensible User Interface Language (XUL)	The Mozilla project created XUL for describing graphical user interfaces in a platform-independent way. For more information visit: **`www.mozilla.org/xpfe/languageSpec.html`**.

Fig. 15.18 XML Vocabularies.

7. The example in this section requires msxml 3.0 or higher to run. For more information on downloading and installing msxml 3.0, visit **`www.deitel.com`**.
8. XHTML is the W3C Recommendation that replaces HTML for marking up content for the Web. For more information on XHTML, see the XHTML Appendices I and J.

To perform transformations, an XSLT processor is required. Popular XSLT processors include Microsoft's msxml, the Apache Software Foundation's *Xalan 2* and the Python package ***4XSLT*** (which we use in Chapter 16, Python XML Processing). The XML document in Fig. 15.19 is transformed by msxml into an XHTML document using the XSLT document in Fig. 15.20.

Line 6 is a processing instruction specific to IE that specifies the location of the XSLT document to apply to this XML document. Figure 15.20 presents the XSLT document (**`sorting.xsl`**) that transforms **`sorting.xml`** (Fig. 15.19) to XHTML.

Performance Tip 15.5

Using Internet Explorer on the client to process XSLT documents conserves server resources by using the client's processing power (instead of having the server process XSLT documents for multiple clients).

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.19: sorting.xml                      -->
4  <!-- XML document containing book information. -->
5
6  <?xml:stylesheet type = "text/xsl" href = "sorting.xsl"?>
7
8  <book isbn = "999-99999-9-X">
9     <title>Mary&apos;s XML Primer</title>
10
11    <author>
12       <firstName>Mary</firstName>
13       <lastName>White</lastName>
14    </author>
15
16    <chapters>
17       <frontMatter>
18          <preface pages = "2" />
19          <contents pages = "5" />
20          <illustrations pages = "4" />
21       </frontMatter>
22
23       <chapter number = "3" pages = "44">
24          Advanced XML</chapter>
25       <chapter number = "2" pages = "35">
26          Intermediate XML</chapter>
27       <appendix number = "B" pages = "26">
28          Parsers and Tools</appendix>
29       <appendix number = "A" pages = "7">
30          Entities</appendix>
31       <chapter number = "1" pages = "28">
32          XML Fundamentals</chapter>
33    </chapters>
34
35    <media type = "CD" />
36 </book>
```

Fig. 15.19 XML document containing book information.

Line 1 of Fig. 15.20 contains the XML declaration. This line is present because an XSLT document is an XML document. Line 6 is the ***xsl:stylesheet*** root element. Attribute ***version*** specifies the version of XSLT to which this document conforms. Namespace prefix **xsl** is defined and bound to the XSLT URI defined by the W3C. When processed, lines 11–13 write the document type declaration to the result tree. Attribute ***method*** is assigned ***"xml"***, which indicates that XML is being output to the result tree. Attribute ***omit-xml-declaration*** is assigned **"no"**, which indicates that an XML declaration will be output to the result tree. Attribute ***doctype-system*** and ***doctype-public*** contain the **Doctype** DTD information that is output to the result tree.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.20: sorting.xsl                           -->
4  <!-- Transformation of book information into XHTML. -->
5
6  <xsl:stylesheet version = "1.0"
7     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9     <!-- write XML declaration and DOCTYPE DTD information -->
10    <xsl:output method = "xml" omit-xml-declaration = "no"
11       doctype-system =
12          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
13       doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
14
15    <!-- match document root -->
16    <xsl:template match = "/">
17       <html xmlns = "http://www.w3.org/1999/xhtml">
18          <xsl:apply-templates />
19       </html>
20    </xsl:template>
21
22    <!-- match book -->
23    <xsl:template match = "book">
24       <head>
25          <title>ISBN <xsl:value-of select = "@isbn" /> -
26             <xsl:value-of select = "title" /></title>
27       </head>
28
29       <body>
30          <h1 style = "color: blue">
31             <xsl:value-of select = "title"/></h1>
32
33          <h2 style = "color: blue">by <xsl:value-of
34             select = "author/lastName" />,
35             <xsl:value-of select = "author/firstName" /></h2>
36
37          <table style =
38             "border-style: groove; background-color: wheat">
```

Fig. 15.20 XSLT document that transforms **sorting.xml** into XHTML. (Part 1 of 3.)

```

            <xsl:for-each select = "chapters/frontMatter/*">
               <tr>
                  <td style = "text-align: right">
                     <xsl:value-of select = "name()" />
                  </td>

                  <td>
                    ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/chapter">
               <xsl:sort select = "@number" data-type = "number"
                    order = "ascending" />
               <tr>
                  <td style = "text-align: right">
                     Chapter <xsl:value-of select = "@number" />
                  </td>

                  <td>
                     ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/appendix">
               <xsl:sort select = "@number" data-type = "text"
                    order = "ascending" />
               <tr>
                  <td style = "text-align: right">
                     Appendix <xsl:value-of select = "@number" />
                  </td>

                  <td>
                     ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>
         </table>

         <p style = "color: blue">Pages:
            <xsl:variable name = "pagecount"
               select = "sum(chapters//*/@pages)" />
            <xsl:value-of select = "$pagecount" />
         <br />Media Type:
            <xsl:value-of select = "media/@type" /></p>
      </body>
   </xsl:template>

</xsl:stylesheet>
```

Fig. 15.20 XSLT document that transforms `sorting.xml` into XHTML. (Part 2 of 3.)

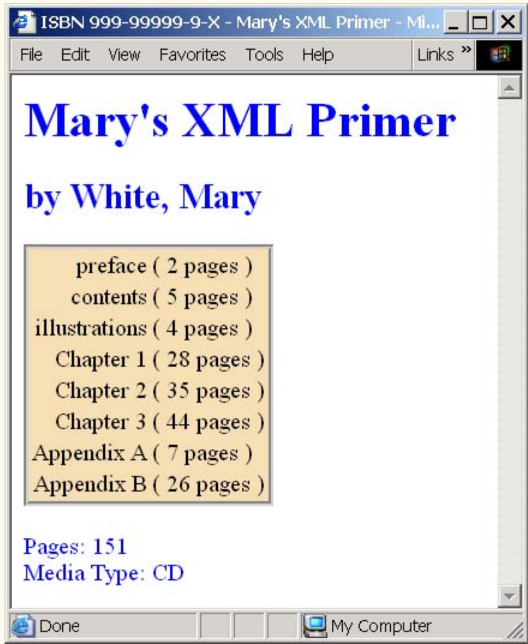


Fig. 15.20 XSLT document that transforms **`sorting.xml`** into XHTML. (Part 3 of 3.)

XSLT documents contain one or more ***`xsl:template`*** elements that specify which information the XSLT processor outputs to the result tree. The template on line 16 ***`match`***es the source tree's document root. When the document root is encountered during the transformation, this **`template`** is applied, and any text marked up by this element that is not in the namespace referenced by **`xsl`** is outputted to the result tree. Line 18 calls for all the **`template`**s that **`match`** children of the document root to be applied. Line 23 specifies a **`template`** that **`match`**es element **`book`**.

Lines 25–26 create the title for the XHTML document. We use the ISBN of the book from attribute **`isbn`** and the contents of element **`title`** to create the title string **ISBN 999-99999-9-X - Mary's XML Primer**. Element **`xsl:value-of`** selects the **`book`** element's **`isbn`** attribute.

Lines 33–35 create a header element that contains the book's author. Because the *context node* (i.e., the current node being processed) is **`book`**, the expression **`author/lastName`** selects the author's last name, and the expression **`author/firstName`** selects the author's first name.

Line 40 selects each element (indicated by an asterisk) that is a child of element **`frontMatter`**. Line 43 calls *node-set function* ***`name`*** to retrieve the current node's element name (e.g., **`preface`**). The current node is the context node specified in the **`xsl:for-each`** (line 40).

Lines 53–54 sort **`chapter`**s by number in ascending order. Attribute **`select`** selects the value of context node **`chapter`**'s attribute **`number`**. Attribute ***`data-type`*** with value **`"number"`**, specifies a numeric sort and attribute ***`order`*** specifies **`"ascending"`**

order. Attribute **`data-type`** also can be assigned the value ***`"text"`*** (line 67) and attribute **`order`** also may be assigned the value ***`"descending"`***.

Lines 82–83 use an *XSLT variable* to store the value of the book's page count and output it to the result tree. Attribute ***`name`*** specifies the variable's name, and attribute ***`select`*** assigns it a value. Function ***`sum`*** totals the values for all **`page`** attribute values. The two slashes between **`chapters`** and **`*`** indicate that all descendent nodes of **`chapters`** are searched for elements that contain an attribute named **`pages`**.

Figure 15.21 shows the XHTML that is generated when msxml applies **`sorting.xsl`** to **`sorting.xml`**. In Chapter 16, we use several of Python's XML-related packages to apply XSLT style sheets to XML documents.

Notice that the XHTML document contains an XML declaration that is different than what was shown previously.Value ***`encoding`*** indicates the type of *character encoding* (i.e., a set of numeric values associated with characters) the document uses. This document uses UTF-8, which is well suited for ASCII-based systems. UTF-8 is the default encoding for XML documents. More information on character encoding and UTF-8 may be found in Appendix F, Unicode.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

   <head>
      <title>ISBN 999-99999-9-X - Mary's XML Primer</title>
   </head>

   <body>
      <h1 style="color: blue">Mary's XML Primer</h1>
      <h2 style="color: blue">by White, Mary</h2>
     <table style="border-style: groove; background-color: wheat">

         <tr>
            <td style="text-align: right">preface</td>
            <td>( 2 pages )</td>
         </tr>

         <tr>
            <td style="text-align: right">contents</td>
            <td>( 5 pages )</td>
         </tr>

         <tr>
            <td style="text-align: right">illustrations</td>
            <td>( 4 pages )</td>
         </tr>

         <tr>
            <td style="text-align: right">Chapter 1</td>
```

Fig. 15.21 XHTML generated when **`sorting.xsl`** is applied by msxml to **`sorting.xml`**. (Part 1 of 2.)

```
            <td>( 28 pages )</td>
         </tr>

         <tr>
            <td style="text-align: right">Chapter 2</td>
            <td>( 35 pages )</td>
         </tr>

         <tr>
            <td style="text-align: right">Chapter 3</td>
            <td>( 44 pages )</td>
         </tr>

         <tr>
            <td style="text-align: right">Appendix A</td>
            <td>( 7 pages )</td>
         </tr>

         <tr>
            <td style="text-align: right">Appendix B</td>
            <td>( 26 pages )</td>
         </tr>

      </table>

      <p style="color: blue">Pages: 11<br />Media Type: CD</p>

   </body>
</html>
```

Fig. 15.21 XHTML generated when **sorting.xsl** is applied by msxml to **sorting.xml**. (Part 2 of 2.)

15.9 Internet and World Wide Web Resources

www.w3.org/xml
The W3C (World Wide Web Consortium) works to develop common protocols to ensure interoperability on the Web. Their XML page includes information about upcoming events, publications, software and discussion groups. Visit this site to read about the latest developments in XML.

www.xml.org
xml.org is a reference for XML, DTDs, Schemas and namespaces. This site also contains news on how XML relates to industry.

www.w3.org/style/XSL
This site provides information on XSL, including what is new in XSL, learning XSL, XSL-enabled tools, the XSL specification, FAQs and the history of XSL.

www.w3.org/TR
This W3C technical reports and publications page contains links to working drafts, proposed recommendations, recommendations and so on.

xml.apache.org
The Apache XML Web site provides many resources related to XML, which include tools and downloads.

www.xmlbooks.com
This site contains a list of XML books recommended by Charles Goldfarb—one of the original designers of GML (General Markup Language), from which XML's parent language SGML (Standard Generalized Markup Language) was derived.

wdvl.internet.com/Authoring/Languages/XML
The *Web Developer's Virtual Library XML* site includes tutorials, FAQ, the latest news and extensive links to XML sites and software downloads.

www.xml.com
Visit **xml.com** for the latest news and information about XML, conference listings, links to XML Web resources organized by topic, tools and more.

msdn.microsoft.com/xml/default.asp
The *MSDN Online XML Development Center* features articles on XML, Ask the Experts chat sessions, samples and demos, newsgroups and other helpful information.

www.oasis-open.org/cover/xml.html
The *SGML/XML Web Page* is an extensive resource that includes links to FAQs, online resources, industry initiatives, demos, conferences and tutorials.

www.gca.org/whats_xml/default.htm
The GCA site has an XML glossary, list of books, brief descriptions of the draft standards for XML and links to online drafts.

www.xmlinfo.com
XMLINFO is a resource site with tutorials, a list of recommended books, documentation, discussion forums and more.

developer.netscape.com/tech/xml/index.html
The *XML and Metadata Developer Central* site has demos, technical notes and news articles related to XML.

www.ucc.ie/xml
This site is a detailed XML FAQ. Submit your own questions through the site.

www.xml-cml.org
This site is a resource for the Chemical Markup Language (CML). It includes a FAQ list, documentation, software and XML links.

SUMMARY

- XML is a widely supported open technology (i.e., nonproprietary technology) for data exchange.
- XML permits document authors to create their own markup for virtually any type of information. This extensibility enables document authors to create entirely new markup languages (called vocabularies) to describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes.
- XML allows document authors to create their own tags, so naming collisions (i.e., different elements that have the same name) can occur. Namespaces enable document authors to prevent collisions among elements in an XML document.
- Namespace prefixes prepended to tag names specify the namespace in which the element can be found. Each namespace prefix has a corresponding uniform resource identifier (URI) that uniquely identifies the namespace. By definition, a URI is a series of characters that differentiates names. Document authors can create their own namespace prefixes. Document authors can use virtually any namespace prefix except the reserved namespace prefix **xml**.

- To eliminate the need to place a namespace prefix in each element, authors may specify a default namespace for an element and all of its child elements.
- XML documents are highly portable. Opening an XML document does not require special software—any text editor that supports ASCII/Unicode characters will suffice. One important characteristic of XML is that it is both human readable and machine readable.
- Processing an XML document—which typically ends in the **.xml** extension—requires a software program called an XML parser (or an XML processor). Parsers check an XML document's syntax and can support the Document Object Model (DOM) and/or the Simple API for XML (SAX) API.
- DOM-based parsers build a tree structure containing the XML document's data in memory. This allows programs to manipulate the document's data. SAX-based parsers process the document and generate events as the parser encounters tags, text, comments and so on. These events contain data from the XML document.
- An XML document can reference an optional document that defines the XML document's structure. This optional document can be either a Document Type Definition (DTD) or a Schema.
- A DOM tree has a single root node that contains all other nodes in the document. The XML parser exposes these methods and properties as a programmatic library, called an Application Programming Interface (API).
- A node that contains other nodes (called child nodes) is a parent node. Nodes that are peers are sibling nodes. A node's descendant nodes include that node's children, its children's children and so on. A node's ancestor nodes include that node's parent, its parent's parent and so on.
- If the XML document conforms to its DTD or Schema, then the XML document is valid. Parsers that cannot check for document conformity against DTDs or Schemas are called nonvalidating parsers. If an XML parser (validating or nonvalidating) can process an XML document that does not have a DTD or Schema successfully, the XML document is well formed (i.e., it is syntactically correct). By definition, a valid XML document also is a well-formed document.
- The **ATTLIST** element type declaration in a DTD defines an attribute. Keyword **#IMPLIED** specifies that, if the parser finds an element without the attribute, the application can provide a value or ignore the missing attribute. Keyword **#REQUIRED** specifies that the attribute must be in the document, and keyword **#FIXED** specifies that the attribute must have the given value. Flag **CDATA** specifies that an attribute contains data that the parser should not process as markup. Keyword **EMPTY** specifies that the element does not contain any text.
- Flag **#PCDATA** specifies that the element can store parsed character data (i.e., text). Document authors must replace the characters less than (**<**) and ampersand (**&**) with their corresponding entity references (i.e., **<** and **&**).
- Schemas use XML syntax.
- In XML Schema, element **element** defines an element. Attributes **name** and **type** specify the **element**'s name and data type, respectively. Any element that contains attributes or child elements must define a type—called a complex type—that defines each attribute and child element.
- Attribute **minOccurs** specifies the minimum number of occurrences for an element. Attribute **maxOccurs** specifies the maximum number of occurrences for an element.
- When an element is a simple type, such as **xsd:string**, that element cannot contain attributes and child elements.
- MathML markup describes mathematical expressions.
- Chemical Markup Language (CML) marks up molecular and chemical information.
- The characters **<?** and **?>** delimit processing instructions (PIs), which are application-specific information embedded in an XML document. A processing instruction consists of a PI target and a PI value.

- Extensible Stylesheet Language (XSL) documents specify how programs should render an XML document's data. A subset of XSL—XSL Transformations (XSLT)—provides elements that define rules for transforming data from one XML document into another text-based format such as XHTML.
- Transforming an XML document using XSLT involves two tree structures: The source tree (i.e., the XML document being transformed) and the result tree (i.e., the XML document to create).

TERMINOLOGY

ancestor node
asterisk (`*`) occurrence indicator
`atomArray` element
`ATTLIST` element type declaration
`CDATA` flag
child node
`complexType` element
container element
context node
`data-type` attribute
default namespace
descendent node
`doctype-public` attribute
`doctype-system` attribute
document reuse
document root
Document Type Definition (DTD)
DOM (Document Object Model)
DOM API (Application Programming Interface)
DOM-based XML parser
EBNF (Extended Backus-Naur Form) grammar
`ELEMENT` element type declaration
empty element
`EMPTY` keyword
event
Extensible Stylesheet Language (XSL)
external DTD
forward slash
`#IMPLIED` flag
invalid document
`match` attribute
`maxOccurs` attribute
`minOccurs` attribute
`mn` element
`molecule` element
`mrow` element
`msqrt` element
`msub` element
`msubsup` element
msxml parser
`name` attribute
`name` node-set function
namespace prefix
node
nonvalidating XML parser
occurrence indicator
`order` attribute
parent node
parsed character data
parser
`#PCDATA` flag
PI (processing instruction)
PI target
PI value
plus sign (`+`) occurrence indicator
processing instruction
question mark (`?`) occurrence indicator
result tree
root element
root node
SAX (Simple API for XML)
SAX-based parser
`schema` element
Schema valid
`select` attribute
simple type
single-quote character (`'`)
source tree
`stylesheet` element
`sum` function
`SYSTEM` flag
`targetNamespace` attribute
tree-based model
`type` attribute
`unbounded` value
validating XML parser
well-formed document
XML (Extensible Markup Language)
XML declaration
`.xml` file extension
`xml` namespace prefix
XML parser
XML processor
XML Schema

XML **version**
xmlns keyword
.xsd extension
XSL (Extensible Stylesheet Language)
.xsl extension
XSL Transformations (XSLT)
xsl:apply-templates element
xsl:for-each element
xsl:output element
xsl:sort element
xsl:value-of element
XSLT variable
XSV (XML Schema Validator)

SELF-REVIEW EXERCISES

15.1 Which of the following tag names might be found in a well-formed XML document?

a) **yearBorn**.
b) **year.Born**.
c) **year Born**.
d) **year-Born1**.
e) **2_year_born**.
f) **--year/born**.
g) **year*born**.
h) **.year_born**.
i) **_year_born_**.
j) **y_e-a_r-b_o-r_n**.

15.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) XML is a technology for creating markup languages.
b) Forward and backward slashes (**/** and ****) delimit XML markup text.
c) All XML start tags must have corresponding end tags.
d) Parsers check an XML document's syntax.
e) **XML**, in any mixture of case, is a reserved namespace prefix.
f) When creating XML documents, document authors must use the set of XML tags that the W3C provides.
g) In an XML document, the pound character (**#**), the dollar sign (**$**), ampersand (**&**), greater-than (**>**) and less-than (**<**) must be replaced with their corresponding entity references.

15.3 Fill in the blanks for each of the following statements:

a) MathML element _________ defines a mathematical operator.
b) _________ help avoid naming collisions.
c) _________ embed application-specific information into an XML document.
d) _________ is Microsoft's XML parser.
e) XSL element _________ inserts a **DOCTYPE** in the result tree.
f) XML Schema documents have root element _________.
g) Element _________ marks up the **∫** MathML entity reference.
h) _________ defines attributes in a DTD.
i) XSL element _________ is the root element in an XSL document.
j) XSL element _________ selects specific XML elements using repetition.

15.4 State whether each of the following is *true* or *false*. If *false*, explain why.

a) XML is not case sensitive.
b) An XML document may contain only one root element.
c) XML is a formatting language.
d) A DTD/Schema defines the style of an XML document.
e) MathML is an XML vocabulary.
f) XSL is an acronym for XML Stylesheet Language.

g) The **<!ELEMENT list (item*)>** defines element **list** as containing one or more **item** elements.

h) XML documents must have the **.xml** extension.

15.5 Find the error(s) in each of the following and explain how to correct it (them).

a)
```
<job>
   <title>Manager</title>
   <task number = "42">
</job>
```

b)
```
<mfrac>
   <mi>x</mi>
   <mo>+</mo>
   <mn>4</mn>
   <mi>y</mi>
</mfrac>
```

c)
```
<company name = "Deitel & Associates, Inc." />
```

15.6 What is the **#PCDATA** flag used for?

15.7 Write a processing instruction for Internet Explorer that includes the style sheet **wap.xsl**.

ANSWERS TO SELF-REVIEW EXERCISES

15.1 a, b, d, i, j. [Choice c is incorrect because it contains a space; Choice e is incorrect because the first character is a number; Choice f is incorrect because it contains a division symbol (**/**) and does not begin with a letter or underscore; Choice g is incorrect because it contains an asterisk (*****); Choice h is incorrect because the first character is a period (**.**) and does not begin with a letter or underscore.]

15.2 a) True. b) False. In an XML document, markup text is delimited by angle brackets (**<** and **>**) with a forward slash being used in the end tag. c) True. d) True. e) True. f) False. When creating tags, document authors may use any permissible name except the reserved word **xml** in any mixture of case. g) False. The ampersand (**&**) and the left-angle bracket (**<**) must be replaced with their entity references.

15.3 a) **mo**. b) namespaces. c) processing instructions. d) msxml. e) **xsl:output**. f) **Schema**. g) **mo**. h) **ATTLIST**. i) **xsl:stylesheet**. j) **xsl:for-each**.

15.4 a) False. XML is case sensitive. b) True. c) False. XML organizes data in a structured manner. d) False. A DTD/Schema defines an XML document's structure. e) True. f) False. XSL is an acronym for Extensible Stylesheet Language. g) False. Element **list** can contain any number of optional **item** elements. h) False. An XML document can have any extension.

15.5 a) The closing **/** in empty element **task** is missing:
```
<task number = "42"/>
```
b) **<mrow>** tag is needed to contain *x* + *4*.

c) The ampersand must be replaced with **&**:
```
<company name = "Deitel &amp; Associates, Inc." />
```

15.6 Flag **#PCDATA** denotes that parsed character data is contained in the element.

15.7
```
<?xsl:stylesheet type = "text/xsl" href = "wap.xsl"?>
```

EXERCISES

15.8 Create an XML document that marks up the nutrition facts for a package of Grandma Deitel's Cookies. A package of Grandma Deitel's Cookies has a serving size of 1 package and the following nutritional value per serving: 260 calories, 100 fat calories, 11 grams of fat, 2 grams of saturated fat,

5 milligrams of cholesterol, 210 milligrams of sodium, 36 grams of total carbohydrates, 2 grams of fiber, 15 grams of sugar and 5 grams of protein. Load the XML document in Internet Explorer. [*Hint*: Your markup should contain elements that describe the product name, serving size/amount, calories, sodium, cholesterol, protein, etc. Mark up each nutrition fact/ingredient listed above.]

15.9 Write an XSLT style sheet for your solution to Exercise 15.8 that displays the nutritional facts in an XHTML table.

15.10 Write an XML document that marks up the following information in Fig. 15.22.

15.11 Write a DTD for the XML document in Exercise 15.10.

15.12 Modify your solution to Exercise 15.10 to qualify each person with a namespace prefix corresponding to their job. Your solution should not contain any elements or attributes that identify a person's job.

15.13 Write an XSLT document that transforms the XML document of Exercise 15.10 into an XHTML sorted table.

15.14 Modify Fig. 15.20 (**`sorting.xsl`**) to sort each section (i.e., front matter, chapters and appendix) of the book by page number, rather than by section.

Name	Job	Department	Cubicle
Joe	Programmer	Engineering	5E
Erin	Designer	Marketing	9M
Melissa	Designer	Human Resources	8H
Craig	Administrator	Engineering	4E
Eileen	Project Coordinator	Marketing	3M
Danielle	Programmer	Engineering	12E
Frank	Salesperson	Marketing	17M
Corinne	Programmer	Technical Support	19T

Fig. 15.22 Information for Exercise 15.10.

16

Python XML Processing

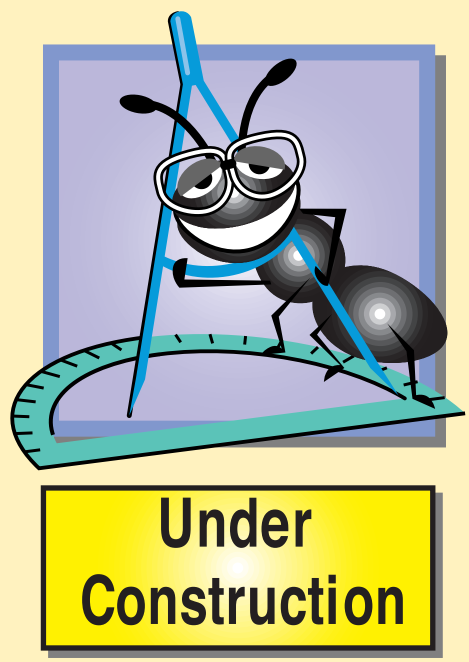
Objectives

- To create XML markup programmatically.
- To use the Document Object Model (DOM™) to manipulate XML documents.
- To use the Simple API for XML (SAX) to retrieve data from XML documents.
- To create an XML-based message forum.

Knowing trees, I understand the meaning of patience.
Knowing grass, I can appreciate persistence.
Hal Borland

I think that I shall never see
A poem lovely as a tree.
Joyce Kilmer

I played with an idea, and grew willful; tossed it into the air; transformed it; let it escape and recaptured it; made it iridescent with fancy, and winged it with paradox.
Oscar Wilde



Outline

16.1 Introduction

In Chapter 15, we introduced XML and various XML-related technologies. In this chapter, we demonstrate how Python applications and scripts can process XML documents. Support for XML is provided through a large collection of freely available Python packages and modules. This chapter focuses on the two of these Python packages: **`4DOM`** and **`xml.sax`**.

In this chapter, we discuss how to generate XML content programatically. We introduce DOM- and SAX-based parsing for programmatically manipulating an XML document's data. The chapter concludes with a case study that uses XML to mark up an online message forum's data.

16.2 Generating XML Content Dynamically

The process by which Python applications can generate XML dynamically is similar to the process by which they generate XHTML. For example, to output XML from a Python script, we can use **`print`** statements or we can use XSLT.

In this section, we present a simple Python script that creates an XML document from data in a text file (Fig. 16.1). The XML markup is sent to the browser via **`print`** statements. In Section 16.4, we present more sophisticated techniques for creating and manipulating XML documents. Figure 16.2 is the Python script that marks up the text file's data as XML. [*Note*: Files **`names.txt`**, **`fig16_02.py`** and **`contact_list.xsl`** must be placed in the correct directories for this example to be served by Apache. Specifically, **`names.txt`** and **`fig16_02.py`** must be located in Apache's **`cgi-bin`** directory. **`contact_list.xsl`** must be located in a directory called **`XML`** under Apache's **`htdocs`** directory. The correct directory structure can also be seen in Fig. 16.19.]

```
O'Black, John
Green, Sue
```

Fig. 16.1 Text file **`names.txt`** used in Fig. 16.2. (Part 1 of 2.)

```
Red, Bob
Blue, Mary
White, Mike
Brown, Jane
Gray, Bill
```

Fig. 16.1 Text file **names.txt** used in Fig. 16.2. (Part 2 of 2.)

```
#!c:\Python\python.exe
# Fig. 16.2: fig16_02.py
# Marking up a text file's data as XML.

import sys

print "Content-type: text/xml\n"

# write XML declaration and processing instruction
print """<?xml version = "1.0"?>
<?xml:stylesheet type = "text/xsl"
   href = "../XML/contact_list.xsl"?>"""

# open data file
try:
   file = open( "names.txt", "r" )
except IOError:
   sys.exit( "Error opening file" )

print "<contacts>" # write root element

# list of tuples: ( special character, entity reference )
replaceList = [ ( "&", "&amp;" ),
                ( "<", "&lt;" ),
                ( ">", "&gt;" ),
                ( '"', "&quot;" ),
                ( "'", "&apos;" ) ]

# replace special characters with entity references
for currentLine in file.readlines():

   for oldValue, newValue in replaceList:
      currentLine = currentLine.replace( oldValue, newValue )

   # extract lastname and firstname
   last, first = currentLine.split( ", " )
   first = first.strip() # remove carriage return

   # write contact element
   print """   <contact>"
      <LastName>%s</LastName>
      <FirstName>%s</FirstName>
   </contact>""" % ( last, first )

```

Fig. 16.2 Marking up a text file's data as XML. (Part 1 of 2.)

```
45 file.close()
46
47 print "</contacts>"
```

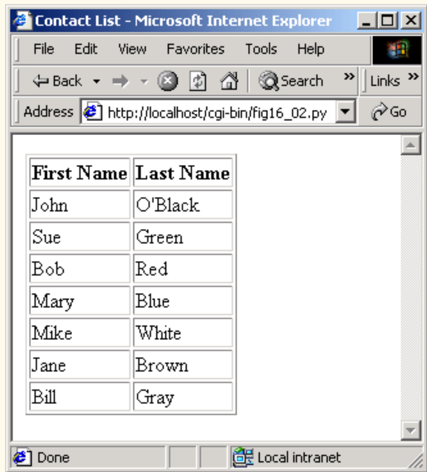


Fig. 16.2 Marking up a text file's data as XML. (Part 2 of 2.)

Line 7 prints the HTTP header, which sets the MIME type to **`text/xml`**. Lines 10–12 print the XML declaration and a processing instruction for Internet Explorer. The processing instruction references the XSLT style sheet **`contact_list.xsl`** (Fig. 16.3).

After the script prints the headers, lines 15–18 open the file (or exit, if the file could not be opened). Line 20 prints the **`<contacts>`** start tag of the root element. A list of five tuples is created in lines 23–27. Each tuple contains two values: a character and an entity reference that corresponds to that character. The **`for`** loop in lines 30–43 generates XML elements for each name in the file. Lines 32–33 call method **`replace`** to substitute characters (e.g., **`<`**, **`&`**, etc.) with their corresponding entity references. The **`split`** method (line 36) extracts the last name and first name from the line read from the file. Line 37 removes any whitespace (e.g., a carriage return) from the first name. The XML element containing the person's name is printed in lines 40–43. Finally, line 47 prints the root element's end tag.

```
<?xml version = "1.0"?>
<!-- Fig. 16.3: contact_list.xsl -->
<!-- Formats a contact list      -->

<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <!-- match document root -->
   <xsl:template match = "/">

      <html xmlns = "http://www.w3.org/1999/xhtml">

```

Fig. 16.3 XSLT used to format contact list. (Part 1 of 2.)

```
         <head>
            <title>Contact List</title>
         </head>

         <body>
            <table border = "1">

               <thead>
                  <tr>
                     <th>First Name</th>
                     <th>Last Name</th>
                  </tr>
               </thead>

               <!-- process each contact element -->
               <xsl:for-each select = "contacts/contact">
                  <tr>
                     <td>
                        <xsl:value-of select = "FirstName" />
                     </td>
                     <td>
                        <xsl:value-of select = "LastName" />
                     </td>
                  </tr>
               </xsl:for-each>

            </table>

         </body>

      </html>

   </xsl:template>

</xsl:stylesheet>
```

Fig. 16.3 XSLT used to format contact list. (Part 2 of 2.)

16.3 XML Processing Packages

In the remaining sections of this chapter, we provide several examples of XML processing using the Document Object Model (DOM) and Simple API for XML (SAX). At the time of this writing, the modules included with Python for DOM manipulation were ***xml.minidom*** and ***xml.pulldom***. Neither of these DOM implementations is fully compliant with the W3C's DOM Recommendation. Therefore, we use a third-party package called ***4DOM***, which fully complies with the W3C's DOM Recommendation. **4DOM** is included with the package ***PyXML***[1] (**pyxml.sourceforge.net**). The classes and functions provided by **4DOM** are located in **xml.dom.ext**.

In Section 16.5, we use a package that is included with Python[2]—***xml.sax***—that contains classes and functions for SAX-based parsing.

1. Visit **www.deitel.com** for installation instructions.
2. Version 2.0 and higher.

Another package, ***4XSLT***, contains an XSLT processor for transforming XML documents into other text-based formats. **4XSLT** is located in a package called ***4Suite***[3] (**4suite.org**), from Fourthought, Inc. The classes and functions provided by **4XSLT** are located in **xml.xslt**.

16.4 Document Object Model (DOM)

In Chapter 15, we introduced the Document Object Model (DOM). In this section, we demonstrate how to use Python and the DOM API to manipulate XML documents programatically.

Figure 16.4 takes an XML document (Fig. 16.5) that marks up an article and uses the DOM implementation included in **4DOM** to display the document's element names and values.

```
# Fig. 16.4: fig16_04.py
# Using 4DOM to traverse an XML Document.

import sys
from xml.dom.ext import StripXml
from xml.dom.ext.reader import PyExpat
from xml.parsers.expat import ExpatError

# open XML file
try:
   file = open( "article2.xml" )
except IOError:
   sys.exit( "Error opening file" )

# parse contents of XML file
try:
   reader = PyExpat.Reader()             # create Reader instance
   document = reader.fromStream( file ) # parse XML document
   file.close()
except ExpatError:
   sys.exit( "Error processing XML file" )

# get root element
rootElement = StripXml( document.documentElement )
print "Here is the root element of the document: %s" % \
   rootElement.nodeName

# traverse all child nodes of root element
print "The following are its child elements:"

for node in rootElement.childNodes:
   print node.nodeName

```

Fig. 16.4 Traversing an XML document. (Part 1 of 2.)

3. **PyXML** must be installed prior to installing **4Suite**. Visit **www.deitel.com** for installation instructions.

```
34   # get first child node of root element
35   child = rootElement.firstChild
36   print "\nThe first child of root element is:", child.nodeName
37   print "whose next sibling is:",
38
39   # get next sibling of first child
40   sibling = child.nextSibling
41   print sibling.nodeName
42   print 'Value of "%s" is:' % sibling.nodeName,
43
44   value = sibling.firstChild
45
46   # print text value of sibling
47   print value.nodeValue
48   print "Parent node of %s is: %s" % \
49      ( sibling.nodeName, sibling.parentNode.nodeName )
50
51   reader.releaseNode( document ) # remove DOM tree from memory
```

```
Here is the root element of the document: article
The following are its child elements:
title
date
author
summary
content

The first child of root element is: title
whose next sibling is: date
Value of "date" is: December 19, 2001
Parent node of date is: article
```

Fig. 16.4 Traversing an XML document. (Part 2 of 2.)

Lines 10–11 attempt to open **article2.xml** for reading. If the file cannot be opened, the program exits with the message **"Error opening file"** (lines 12–13). Line 17 instantiates a ***PyExpat Reader*** object, which is an instance of a DOM-based parser. Module **PyExpat** is located in **4DOM**'s ***reader*** package. Line 18 passes the XML document referenced by **file** to **Reader** method ***fromStream***, which parses the document and loads the XML document's data into memory. Variable **document** references the DOM tree (called a ***Document***) returned by **fromStream**.

A **Document** object's ***documentElement*** attribute refers to the **Document**'s root element node. Line 24 passes the root element node to **4DOM**'s ***StripXml*** function, which removes insignificant whitespace (e.g., the carriage return line feeds and spaces used for indentation) from an XML DOM tree. If **StripXml** is not called, insignificant whitespace would be stored in the DOM tree. Recall from Chapter 15, that a DOM tree contains a set of nodes. Each node in a DOM tree is of a type derived from class ***Node***. We say more about these derived classes momentarily.

Lines 25–26 print the name of **rootElement** via its ***nodeName*** attribute. A **Node** object's ***childNodes*** attribute is a list of that **Node**'s children. Lines 31–32 print the

nodeName of each child node of **rootElement**. Lines 35–49 then print the names of specific nodes. A **Node** object's ***firstChild*** attribute corresponds to the first child node in that **Node**'s list of children. Lines 35–36 assign the first child of **rootElement** to variable **child** and print the child's name.

Line 40 assigns the next sibling of **child** to variable **sibling**. Attribute ***nextSibling*** contains a node's next sibling (i.e., the next node that has the same parent node). For example, **title**, **date**, **author**, **summary** and **content** are sibling nodes. Line 41 prints **sibling**'s name.

Line 44 assigns the first child node of **sibling** to variable **value**. In this case, **value** is a ***Text*** node that represents the contents of **sibling**. **Text** nodes contain character data. Line 47 prints the text contained in **value** by accessing its ***nodeValue*** attribute. Lines 48–49 print **sibling**'s parent node. Parent nodes are obtained through the ***parentNode*** attribute. Finally, line 51 calls **Reader** method **releaseNode**, which removes the specified **Document** (i.e., DOM tree) from memory.

Good Programming Practice 16.1

*Although not required in Python version 2.0 and higher, calling method **releaseNode** ensures that a DOM tree is freed from memory.*

The classes that inherit from **Node** represent the various XML node types. The **Document** node represents the entire XML document (in memory) and provides methods for manipulating its data. ***Element*** nodes represent XML elements. ***Text*** nodes represent character data. ***Attr*** nodes represent XML attributes, and ***Comment*** nodes represent comments. **Document** nodes can contain **Element**, **Text** and **Comment** nodes. **Element** nodes can contain **Attr**, **Element**, **Text** and **Comment** nodes.

```
<?xml version = "1.0"?>

<!-- Fig. 16.5: article2.xml    -->
<!-- Article formatted with XML -->

<article>

   <title>Simple XML</title>

   <date>December 19, 2001</date>

   <author>
      <firstName>Jane</firstName>
      <lastName>Doe</lastName>
   </author>

   <summary>XML is easy.</summary>

   <content>Once you have mastered XHTML, XML is learned
      easily. Remember that XML is not for displaying
      information but for managing information.
   </content>

</article>
```

Fig. 16.5 XML document used in Fig. 16.4.

The tables in Fig. 16.6–Fig. 16.12 summarize important DOM attributes and methods for navigating and updating DOM trees. Figure 16.6 describes some **Node** attributes and methods, Fig. 16.7 describes some ***NodeList*** (i.e., an ordered list of **Node**s) attributes and methods, Fig. 16.8 describes some ***NamedNodeMap*** (i.e., an unordered dictionary of **Node**s) attributes and methods, Fig. 16.9 describes some **Document** attributes and methods, Fig. 16.10 describes some **Element** attributes and methods, Fig. 16.11 describes some **Attr** attributes and Fig. 16.12 describes a **Text** and **Comment** attribute.

The program in Fig. 16.13 uses the DOM to add names to the contact list XML document, **contacts.xml** (Fig. 16.14). The XML document is loaded into memory, programmatically manipulated and saved to disk (overwriting the previous version).

Attribute/Method	Description
appendChild(*newChild* **)**	Appends *newChild* to the list of child nodes. Returns the appended child node.
attributes	**NamedNodeMap** that contains the attribute nodes for the current node.
childNodes	**NodeList** that contains the node's current children.
firstChild	First child node in the **NodeList** or **None**, if the node has no children.
insertBefore(*newChild*, *refChild* **)**	Inserts the *newChild* node before the *refChild* node. *refChild* must be a child node of the current node; otherwise, **insertBefore** raises a **ValueError** exception.
isSameNode(*other* **)**	Returns true if *other* is the current node.
lastChild	Last child node in the **NodeList** or **None**, if the current node has no children.
nextSibling	The next node in the **NodeList**, or **None**, if the node has no next sibling.
nodeName	Name of the node, or **None**, if the node does not have a name.
nodeType	Integer that represents the node type. Class **Node** defines several constants including: **ELEMENT_NODE** = **1** **ATTRIBUTE_NODE** = **2** **TEXT_NODE** = **3** **COMMENT_NODE** = **8** **DOCUMENT_NODE** = **9**
nodeValue	The current node's value, or **None**, if the node has no value.
parentNode	Parent node or **None** if the node has no parent.

Fig. 16.6 **Node** attributes and methods. (Part 1 of 2.)

Attribute/Method	Description
previousSibling	The previous node in the **NodeList**, or **None**, if the node has no preceding sibling.
removeChild(*oldChild* **)**	Removes a child node. *oldChild* must be a child node of the current node; otherwise, a **ValueError** exception is raised.
replaceChild(*newChild*, *oldChild* **)**	Replaces *oldChild* with *newChild*. *oldChild* must be a child node of the current node; otherwise, **replaceChild** raises a **ValueError** exception.

Fig. 16.6 **Node** attributes and methods. (Part 2 of 2.)

Attribute/Method	Description
item(*i* **)**	Returns the node at index *i*. Indices range from 0 to **length** – 1.
length	Number of nodes in the **NodeList**.

Fig. 16.7 **NodeList** attributes and methods.

Attribute/Method	Description
item(*i* **)**	Returns the attribute node at index *i*. Indices range from 0 to **length** – 1.
length	Number of attribute nodes for the given element node.

Fig. 16.8 **NamedNodeMap** attributes and methods.

Attribute/Method	Description
createAttribute(*name* **)**	Creates and returns an **Attr** node with the specified *name*.
createComment(*data* **)**	Creates and returns a **Comment** node that contains the specified *data*.
createElement(*tagName* **)**	Creates and returns an **Element** node with the specified *tagName*.
createTextNode(*data* **)**	Creates and returns a **Text** node that contains the specified *data*.
documentElement	Root element node of the document tree (DOM tree).

Fig. 16.9 **Document** attributes and methods. (Part 1 of 2.)

Attribute/Method	Description
getElementsByTagName(*name* **)**	Returns a **NodeList** of all nodes in the subtree with the tag name *name*.

Fig. 16.9 **Document** attributes and methods. (Part 2 of 2.)

Attribute/Method	Description
getAttribute(*name* **)**	Returns XML attribute *name*'s value as a string.
getAttributeNode(*name* **)**	Returns the **Attr** node for XML attribute *name*.
getElementsByTagName(*name* **)**	Returns a **NodeList** of all nodes in the subtree with the tag name *name*.
removeAttribute(*name* **)**	Removes XML attribute *name* (specified as a string) from the XML attribute list for the given element node.
removeAttributeNode(*name* **)**	Removes **Attr** node *name* from the XML attribute list for the given **Element** node.
setAttribute(*name*, *value* **)**	Changes the value of XML attribute *name* to *value*. Both arguments are specified as strings.
setAttributeNode(*name* **)**	Adds new **Attr** node *name* to the attribute list for the given element node. If the attribute already exists, the new attribute replaces the current attribute.
tagName	Element's tag name.

Fig. 16.10 **Element** attributes and methods.

Attribute	Description
name	Name of the XML attribute.
prefix	Namespace prefix, if it exists, or **None**.

Fig. 16.11 **Attr** attributes.

Attribute	Description
data	**Node**'s (**Text** or **Comment**) data.

Fig. 16.12 **Text** and **Comment** attribute.

```
# Fig. 16.13: fig16_13.py
# Using 4DOM to manipulate an XML Document.

import sys
from xml.dom.ext.reader import PyExpat
from xml.dom.ext import PrettyPrint

def printInstructions():
   print """\nEnter 'a' to add a contact.
Enter 'l' to list contacts.xml.
Enter 'i' for instructions.
Enter 'q' to quit."""

def printList( document ):
   print "Your contact list is:"

   # iterate over NodeList of contact elements
   for contact in document.getElementsByTagName( "contact" ):
      first = contact.getElementsByTagName( "FirstName" )[ 0 ]

      # get first node's value
      firstText = first.firstChild.nodeValue

      # get NodeList for nodes that contain tag name "LastName"
      last = contact.getElementsByTagName( "LastName" )[ 0 ]
      lastText = last.firstChild.nodeValue

      print firstText, lastText

def addContact( document ):
   root = document.documentElement   # get root element node

   name = raw_input(
      "Enter the name of the person you wish to add: " )

   first, last = name.split()

   # create first name element node
   firstNode = document.createElement( "FirstName" )
   firstNodeText = document.createTextNode( first )
   firstNode.appendChild( firstNodeText )

   # create last name element node
   lastNode = document.createElement( "LastName" )
   lastNodeText = document.createTextNode( last )
   lastNode.appendChild( lastNodeText )

   # create contact node, append first name and last name nodes
   contactNode = document.createElement( "contact" )
   contactNode.appendChild( firstNode )
   contactNode.appendChild( lastNode )

   root.appendChild( contactNode ) # add contact node
```

Fig. 16.13 Manipulating an XML document. (Part 1 of 2.)

```

# open contacts file
try:
   file = open( "contacts.xml", "r+" )
except IOError:
   sys.exit( "Error opening file" )

# create DOM parser and parse XML document
reader = PyExpat.Reader()
document = reader.fromStream( file )

printList( document )
printInstructions()
character = "l"

while character != "q":
   character = raw_input( "\n? " )

   if character == "a":
      addContact( document )
   elif character == "l":
      printList( document )
   elif character == "i":
      printInstructions()
   elif character != "q":
      print "Invalid command!"

file.seek( 0, 0 )                  # position to beginning of file
file.truncate()                    # remove data from file
PrettyPrint( document, file )      # print DOM contents to file
file.close()                       # close XML file
reader.releaseNode( document )     # free memory
```

```
Your contact list is:
John Black
Sue Green

Enter 'a' to add a contact.
Enter 'l' to list contacts.xml.
Enter 'i' for instructions.
Enter 'q' to quit.

? a
Enter the name of the person you wish to add: Michael Red

? l
Your contact list is:
John Black
Sue Green
Michael Red

? q
```

Fig. 16.13 Manipulating an XML document. (Part 2 of 2.)

Line 57 opens **contacts.xml** for reading and writing. A parser object is instantiated on line 62. Line 63 calls method **fromStream** to parse the XML document and build the DOM tree.

Line 65 calls function **printList** (lines 14–28) to print the contact list to the screen. Method ***getElementsByTagName*** (line 18) returns a **NodeList** that contains all **Element** nodes that have **contact** for a tag name. Line 19 calls **getElementsByTagName** to obtain a **NodeList** for all **Element** nodes that have **FirstName** for a tag name. Each node referenced by **contact** contains only one such node. This one node is accessed as the first element in the list (i.e., **[0]**). Line 22 assigns the value of **first**'s first child element (a **Text** node) to variable **firstText**. Lines 25–26 repeat the processes to obtain the last name. Line 28 prints the current contact's first name and last name to the screen.

Line 66 calls function **printInstructions** to print the program's instructions. Lines 69–79 get the user's choice and call the appropriate function.

The **addContact** function (lines 30–53) adds a contact to the list. The **Document**'s root element is obtained via its **documentElement** attribute (line 31). Lines 33–36 prompt the user for input and call string method **split** to separate the first name from the last name.

Line 39 calls the **Document**'s ***createElement*** method to create an **Element** node with the tag name **FirstName**. Lines 40–41 create and append a **Text** node to this **Element** node by calling the ***createTextNode*** and ***appendChild*** methods, respectively. Lines 44–46 create an **Element** node with the tag name **LastName** in a similar manner.

Line 49 creates an **Element** node with the tag name **contact**. Lines 50–51 call method **appendChild** to add the **Element** nodes referenced by **firstNode** and **lastNode** to the node referenced by **contactNode**. Line 53 calls method **appendChild** to add the node referenced by **contactNode** to the node referenced by **root**.

When the user has finished adding names to the contact list, the file is saved. The **seek** method (line 81) positions the file pointer to the beginning of the file and method **truncate** (line 82) deletes the contents of the file. Then, **4DOM**'s ***PrettyPrint*** function writes the updated XML to the file (line 83). Function **PrettyPrint** writes an XML DOM tree's data to a specified output stream (with indentation and carriage returns for readability). Lines 84–85 close the file and release the DOM tree from memory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE contacts>
<contacts>
  <contact>
    <LastName>Black</LastName>
    <FirstName>John</FirstName>
  </contact>
  <contact>
    <LastName>Green</LastName>
    <FirstName>Sue</FirstName>
  </contact>
  <contact>
    <FirstName>Michael</FirstName>
    <LastName>Red</LastName>
  </contact>
</contacts>
```

Fig. 16.14 Contact list output by Fig. 16.13.

16.5 Parsing XML with `xml.sax`

In this section, we discuss the **`xml.sax`** package, which provides a set of modules for SAX-based parsing. With SAX-based parsing, the parser reads the input to identify the XML markup. As the parser encounters markup, the parser calls *event handlers* (i.e., methods). For example, when the parser encounters a start tag, the ***`startElement`*** event handler is called; when the parser encounters character data, the ***`characters`*** event handler is called. Programmers override event handlers to provide specialized processing of the XML. Some common SAX event handlers are shown in Fig. 16.15.

Good Programming Practice 16.2

Review the Python on-line documentation for a complete listing of ***`xml.sax`*** *event handlers. This information can be found at:*
`www.python.org/doc/current/lib/content-handler-objects.html`

Figure 16.16 demonstrates SAX-based parsing. This program allows the user to specify a tag name to search for in an XML document. When the tag name is encountered, the program outputs the element's attribute-value pairs. Methods **`startElement`** and **`endElement`** are overriden to handle the events generated when start tags and end tags are encountered. Figure 16.17 contains the XML document used by this program.

Lines 42–43 obtain the name of the XML document to parse and the tag name to locate. Line 46 invokes **`xml.sax`** function ***`parse`***, which creates a SAX parser object. Function **`parse`**'s first argument is either a Python file object or a filename. The second argument passed to **`parse`** must be an instance of class **`xml.sax.ContentHandler`** (or a derived class of **`ContentHandler`**, such as **`TagInfoHandler`**), which is the main callback handler in **`xml.sax`**. Class **`ContentHandler`** contains the methods (Fig. 16.15) for handling SAX events.

If an error occurs during the opening of the specified **`file`**, an **`IOError`** exception is raised, and line 50 displays an error message. If an error occurs while parsing the file (e.g., if the specified XML document is not well-formed), **`parse`** raises a ***`SAXParseException`*** exception, and line 54 displays an error message.

Event Handler	Description
`characters(` *content* **`)`**	Called when the parser encounters character data. The character data is passed as *content* to the event handler.
`endDocument()`	Called when the parser encounters the end of the document.
`endElement(` *name* **`)`**	Called when the parser encounters an end tag. The tag *name* is passed as an argument to the event handler.
`startDocument()`	Called when the parser encounters the beginning of the document.
`startElement(` *name***`,`** *attrs* **`)`**	Called when the parser encounters a start tag. The tag *name* and its attributes (*attrs*) are passed as arguments to the event handler.

Fig. 16.15 **`xml.sax`** event-handler methods.

```
# Fig. 16.16: fig16_16.py
# Demonstrating SAX-based parsing.

from xml.sax import parse, SAXParseException, ContentHandler

class TagInfoHandler( ContentHandler ):
   """Custom xml.sax.ContentHandler"""

   def __init__( self, tagName ):
      """Initialize ContentHandler and set tag to search for"""

      ContentHandler.__init__( self )
      self.tagName = tagName
      self.depth = 0 # spaces to indent to show structure

   # override startElement handler
   def startElement( self, name, attributes ):
      """An Element has started"""

      # check if this is tag name for which we are searching
      if name == self.tagName:
         print "\n%s<%s> started" % ( " " * self.depth, name )

         self.depth += 3

         print "%sAttributes:" % ( " " * self.depth )

         # check if element has attributes
         for attribute in attributes.getNames():
            print "%s%s = %s" % ( " " * self.depth, attribute,
               attributes.getValue( attribute ) )

   # override endElement handler
   def endElement( self, name ):
      """An Element has ended"""

      if name == self.tagName:
         self.depth -= 3
         print "%s</%s> ended\n" % ( " " * self.depth, name )

def main():
   file = raw_input( "Enter a file to parse: " )
   tagName = raw_input( "Enter tag to search for: " )

   try:
      parse( file, TagInfoHandler( tagName ) )

   # handle exception if unable to open file
   except IOError, message:
      print "Error reading file:", message

```

Fig. 16.16 SAX-based parsing example. (Part 1 of 2.)

```
52       # handle exception parsing file
53       except SAXParseException, message:
54          print "Error parsing file:", message
55
56    if __name__ == "__main__":
57       main()
```

```
Enter a file to parse: boxes.xml
Enter tag to search for: box

<box> started
   Attributes:
   size = big

   <box> started
      Attributes:
      size = medium
   </box> ended


   <box> started
      Attributes:
      type = small

      <box> started
         Attributes:
         type = tiny
      </box> ended

   </box> ended

</box> ended
```

Fig. 16.16 SAX-based parsing example. (Part 2 of 2.)

Our example overrides only two event handlers. Methods **`startElement`** and **`endElement`** are called when start tags and end tags are encountered. Method **`startElement`** (lines 16–31) takes two arguments—the element's tag name as a string and the element's attributes. The attributes are passed as an instance of class ***`AttributesImpl`***, defined in **`xml.sax.reader`**. This class provides a dictionary-like interface to the element's attributes.

Line 21 determines whether the element received from the event contains the tag name that the user specified. If so, line 22 prints the start tag, indented by **`depth`** spaces, and line 24 increments **`depth`** by **`3`** to ensure that the next tag printed indented further.

Lines 29–31 print the element's attributes. The **`for`** loop first obtains the attribute names by invoking the ***`getNames`*** method of **`attributes`**. The loop then prints each attribute name and its corresponding value—obtained by passing the current attribute name to the ***`getValue`*** method of **`attributes`**.

Method **`endElement`** (lines 34–39) executes when an end tag is encountered and receives the end tag's name as an argument. If **`name`** contains the tag name specified by the

user, line 38 decreases the indent by decrementing **depth**. Line 39 prints that the specified end tag was found.

16.6 Case Study: Message Forums with Python and XML[4]

In this section, we use XML and several XML-related technologies to create one of the most popular types of Web sites: A *message forum*. Message forums are “virtual” bulletin boards where users discuss various topics. Common features of message forums include discussion groups, question-and-answer sections and general comments. Many Web sites host message forums. Some popular message forums are

```
groups.yahoo.com
web.eesite.com/forums
groups.google.com
```

Figure 16.18 summarizes the files that comprise the message forum. Figure 16.19 shows the directory structure for Apache running on Windows. Fig. 16.20 illustrates some of the key interactions between the files. The main XHTML page generated by **default.py** displays the list of available message forums, which are stored in the XML document **forums.xml**. Hyperlinks are provided to each XML message forum document and to script **addForum.py**, which adds a forum to **forums.xml** and creates an XML message forum, using the markup in **template.xml** as a starting point.

```
<?xml version = "1.0"?>

<!-- Fig. 16.17: boxes.xml           -->
<!-- XML document used in Fig. 16.16 -->

<boxlist>

  <box size = "big">
     This is the big box.

     <box size = "medium">
        Medium sized box
        <item>Some stuff</item>
        <thing>More stuff</thing>
     </box>

     <parcel />
     <box type = "small">
        smaller stuff
        <box type = "tiny">tiny stuff</box>
     </box>

  </box>

</boxlist>
```

Fig. 16.17 XML document used in Fig. 16.16.

4. The implementation of this message forum requires Internet Explorer 5 or higher, and msxml 3.0 or higher. In Section 16.6.3, we discuss how other client browsers, such as Netscape, may be used.

Each XML document that contains a forum (e.g., **feedback.xml**) is transformed into an XHTML document by applying the XSLT document **formatting.xsl**. The XHTML generated is formatted by applying **site.css**. New messages are posted to a forum by **addPost.py**.

File Name	Description
forums.xml	XML document containing available forum titles and their filenames.
default.py	Main page that provides navigational links to the forums.
template.xml	Template for a message forum document.
addForum.py	Adds a new forum.
feedback.xml	Sample message forum.
formatting.xsl	XSLT document for transforming message forums into XHTML.
addPost.py	Adds a message to a forum.
error.html	Displays an error message.
site.css	Style sheet for formatting XHTML content.
forum.py	Transforms XML documents to HTML on the server for non-Internet Explorer clients.

Fig. 16.18 Message-forum documents.

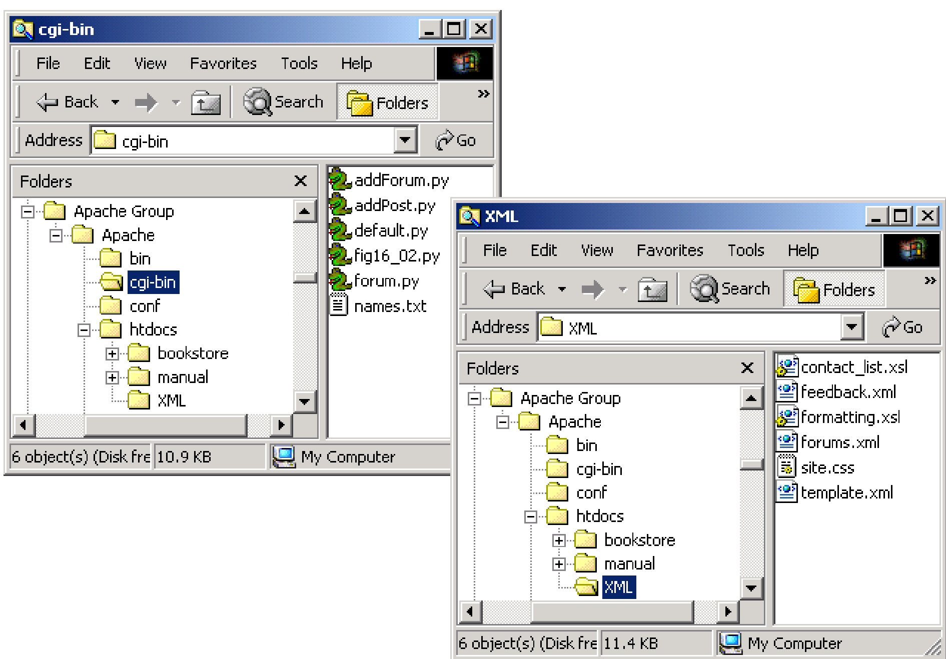


Fig. 16.19 Directory structure for the message forum.

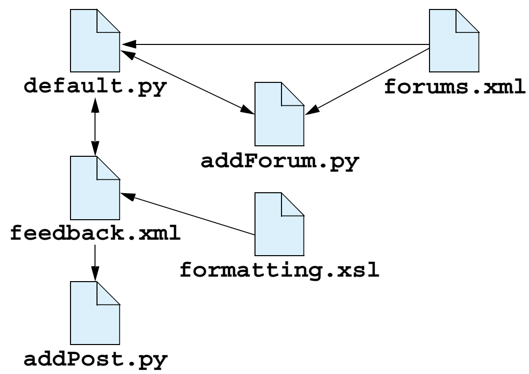


Fig. 16.20 Key interactions between message forum documents.

16.6.1 Displaying the Forums

This section discusses how XML is used to mark up message forum data and the Python script—**`default.py`**—that creates the message forum's main XHTML page. For this case study, we provide a sample forum named **`feedback.xml`** (Fig. 16.21) to show the structure of a forum document.

Notice the reference to the style sheet **`formatting.xsl`** (line 6). When applied by msxml, this XSLT document (which we discuss later in the chapter) transforms the XML into XHTML for display in Internet Explorer. Forum documents have root element **`forum`**, which contains attribute **`file`**. This attribute's value is the document's filename. Child elements include **`name`**, for specifying the title of the forum, and **`message`**, for marking up the message. Messages contain a user name, a title and the text, which are marked up by elements **`user`**, **`title`** and **`text`**, respectively. Messages also are given a **`timestamp`**.

The document **`forums.xml`** (Fig. 16.22) contains the filename and title for every message forum. As forums are created, this document is updated.

Root element **`forums`** (line 8) contains one or more **`forum`** child elements. Initially, one forum (i.e., **`Feedback`**) is present. Each **`forum`** element has attribute **`filename`** and child element **`name`**.

```
1   <?xml version = "1.0"?>
2
3   <!-- Fig. 16.21: feedback.xml         -->
4   <!-- XML document representing a forum -->
5
6   <?xml:stylesheet type = "text/xsl" href = "../XML/formatting.xsl"?>
7
8   <forum file = "feedback.xml">
9     <name>Feedback</name>
10
```

Fig. 16.21 XML document representing a forum containing one message. (Part 1 of 2.)

```
  <message timestamp = "Wed Jun 27 12:53:22 2001">
    <user>Jessica</user>
    <title>Nice forums!</title>
    <text>These forums are great! Well done, all.</text>
  </message>

</forum>
```

Fig. 16.21 XML document representing a forum containing one message. (Part 2 of 2.)

```
<?xml version = "1.0"?>

<!-- Fig. 16.22: forums.xml             -->
<!-- XML document containing all forums -->

<?xml:stylesheet type = "text/xsl" href = "formatting.xsl"?>

<forums>

  <forum filename = "feedback.xml">
    <name>Feedback</name>
  </forum>

</forums>
```

Fig. 16.22 XML document containing data for all available forums.

Visitors to the message forum are greeted initially by the Web page that **default.py** (Fig. 16.23) generates, which displays links to all forums and provides forum management options. Initially, only two links are active—one to view the **Feedback** forum (i.e., the sample forum) and one to create a forum. In the chapter exercises, we ask the reader to enhance the message forum by adding functionality for modifying and deleting forums.

```
#!c:\Python\python.exe
# Fig. 16.23: default.py
# Default page for message forums.

import os
import sys
from xml.dom.ext.reader import PyExpat

def printHeader( title, style ):
   print """Content-type: text/html

<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml">

```

Fig. 16.23 Default page for the message forum. (Part 1 of 3.)

```
<head>
<title>%s</title>
<link rel = "stylesheet" href = "%s" type = "text/css" />
</head>

<body>""" % ( title, style )

# open XML document that contains the forum names and locations
try:
   XMLFile = open( "../htdocs/XML/forums.xml" )
except IOError:
   print "Location: /error.html\n"
   sys.exit()

# parse XML document containing forum information
reader = PyExpat.Reader()
document = reader.fromStream( XMLFile )
XMLFile.close()

# write XHTML to browser
printHeader( "Deitel Message Forums", "/XML/site.css" )
print """<h1>Deitel Message Forums</h1>
<p style="font-weight:bold">Available Forums</p>
<ul>"""

# determine client-browser type
if os.environ[ "HTTP_USER_AGENT" ].find( "MSIE" ) != -1:
   prefix = "../XML/"   # Internet Explorer
else:
   prefix = "forum.py?file="

# add links for each forum
for forum in document.getElementsByTagName( "forum" ):

   # create link to forum
   link = prefix + forum.attributes.item( 0 ).value

   # get element nodes containing tag name "name"
   name = forum.getElementsByTagName( "name" )[ 0 ]

   # get Text node's value
   nameText = name.childNodes[ 0 ].nodeValue
   print '<li><a href = "%s">%s</a></li>' % ( link, nameText )

print """</ul>
<p style="font-weight:bold">Forum Management</p>
<ul>
   <li><a href = "addForum.py">Add a Forum</a></li>
   <li>Delete a Forum</li>
   <li>Modify a Forum</li>
</ul>
</body>
```

Fig. 16.23 Default page for the message forum. (Part 2 of 3.)

```
71   </html>"""
72
73   reader.releaseNode( document )
```

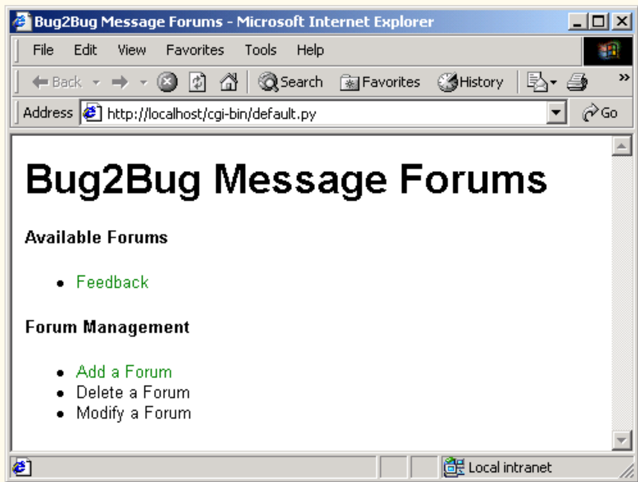


Fig. 16.23 Default page for the message forum. (Part 3 of 3.)

This Python script uses modules in package **`4DOM`** to parse **`forums.xml`**. Lines 33–34 instantiate a parser object, then load and parse **`forums.xml`**. Lines 38–71 output XHTML to the browser. First, line 38 prints the XHTML header for the main page by calling function **`printHeader`** (lines 9–23). This function prints the XHTML header with a specified title and a link to a Cascading Style Sheet (CSS) that formats the page. In this case study, we would like to take advantage of msxml's XML parsing and XSLT processing capabilities to reduce the amount of processing the server must perform. Lines 44–45 determine whether the client is using Internet Explorer. If so, **`prefix`** is set to **`"../XML/"`**. Otherwise, **`prefix`** is set to **`"forum.py?file="`**. Note that line 47 uses **`prefix`** to construct the hyperlinks to each forum. Clients who use Internet Explorer request the XML documents directly, while other clients request **`forum.py`**. We discuss this in greater detail in Section 16.6.3. The **`for`** loop (lines 50–60) retrieves all **`Element`** nodes that contain the tag name **`forum`**. Hyperlinks are created to each forum found in **`forums.xml`**. Lines 62–71 print the remaining XHTML, including a hyperlink to **`addForum.py`**. Finally, line 73 releases the **`Document`** object from memory.

16.6.2 Adding Forums and Messages

In this section, we discuss the Python scripts and documents that add forums and messages. The Python script that adds a new forum is shown in Fig. 16.24. This script uses modules in package **`4DOM`** to manipulate XML documents.

When the script is requested initially, it is not passed any parameters. The script begins by retrieving the form data (line 29). Because the form contains no values, execution begins with the **`else`** block at line 93. Lines 94–107 output a form that prompts the user for a forum name and a filename for the XML document to be created. When the form is sub-

mitted, the script is re-requested and passed the user-entered form values. When this occurs, the condition (line 32) is true, and lines 33–92 execute.

```
1  #!c:\Python\python.exe
2  # Fig. 16.24: addForum.py
3  # Adds a forum to the list
4
5  import re
6  import sys
7  import cgi
8
9  # 4DOM packages
10 from xml.dom.ext.reader import PyExpat
11 from xml.dom.ext import PrettyPrint
12
13 def printHeader( title, style ):
14    print """Content-type: text/html
15
16 <?xml version = "1.0" encoding = "UTF-8"?>
17 <!DOCTYPE html PUBLIC
18    "-//W3C//DTD XHTML 1.0 Strict//EN"
19    "DTD/xhtml1-strict.dtd">
20 <html xmlns = "http://www.w3.org/1999/xhtml">
21
22 <head>
23 <title>%s</title>
24 <link rel = "stylesheet" href = "%s" type = "text/css" />
25 </head>
26
27 <body>""" % ( title, style )
28
29 form = cgi.FieldStorage()
30
31 # if user enters data in form fields
32 if form.has_key( "name" ) and form.has_key( "filename" ):
33    newFile = form[ "filename" ].value
34
35    # determine whether file has xml extension
36    if not re.match( "\w+\.xml$", newFile ):
37       print "Location: /error.html\n"
38       sys.exit()
39    else:
40
41       # create forum files from xml files
42       try:
43          newForumFile = open( "../htdocs/XML/" + newFile, "w" )
44          forumsFile = open( "../htdocs/XML/forums.xml", "r+" )
45          templateFile = open( "../htdocs/XML/template.xml" )
46       except IOError:
47          print "Location: /error.html\n"
48          sys.exit()
49
```

Fig. 16.24 Script that adds a new forum to **forums.xml**. (Part 1 of 3.)

```
      # parse forums document
      reader = PyExpat.Reader()
      document = reader.fromStream( forumsFile )

      # add new forum element
      forum = document.createElement( "forum" )
      forum.setAttribute( "filename", newFile )

      name = document.createElement( "name" )
      nameText = document.createTextNode( form[ "name" ].value )
      name.appendChild( nameText )
      forum.appendChild( name )

      # obtain root element of forum
      documentNode = document.documentElement
      firstForum = documentNode.getElementsByTagName(
         "forum" )[ 0 ]
      documentNode.insertBefore( forum, firstForum )

      # write updated XML to disk
      forumsFile.seek( 0, 0 )
      forumsFile.truncate()
      PrettyPrint( document, forumsFile )
      forumsFile.close()

      # create document for new forum from template file
      document = reader.fromStream( templateFile )
      forum = document.documentElement
      forum.setAttribute( "file", newFile )

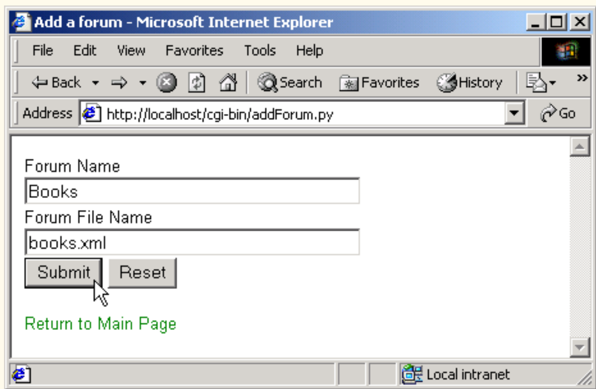
      # create name element
      name = document.createElement( "name" )
      nameText = document.createTextNode( form[ "name" ].value )
      name.appendChild( nameText )
      forum.appendChild( name )

      # write generated XML to new forum file
      PrettyPrint( document, newForumFile )
      newForumFile.close()
      templateFile.close()
      reader.releaseNode( document )

      print "Location: default.py\n"
else:
   printHeader( "Add a forum", "/XML/site.css" )
   print """<form action = "addForum.py" method="post">
Forum Name<br />
<input type = "text" name = "name" size = "40" /><br />
Forum File Name<br />
<input type = "text" name = "filename" size = "40" /><br />
<input type = "submit" name = "submit" value = "Submit" />
<input type = "reset" value = "Reset" />
</form>
```

Fig. 16.24 Script that adds a new forum to **forums.xml**. (Part 2 of 3.)

```
103
104 <a href = "/cgi-bin/default.py">Return to Main Page</a>
105 </body>
106
107 </html>"""
```



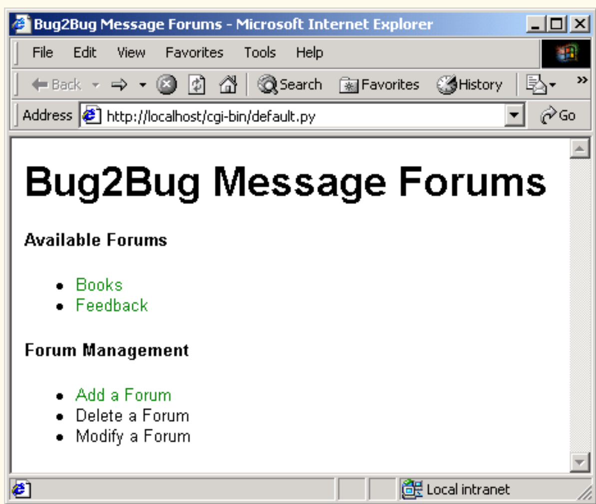


Fig. 16.24 Script that adds a new forum to **forums.xml**. (Part 3 of 3.)

Line 36 examines the filename posted to the script to make sure it contains only alphanumeric characters and ends with **.xml**; if not, the script redirects the client to **error.html**. This prevents a malicious user from writing to a system file or otherwise gaining unrestricted access to the server. However, it is important to note that other solutions exist, such as generating filenames on the server. If the filename is permitted, line 43 attempts to create the file by calling function **open**.

Line 44 opens file **forums.xml** for reading and writing (**"r+"**). Line 40 opens the template XML document named **template.xml** (Fig. 16.25), which provides a forum's

markup. The template contains an empty **forums** element, to which the forum name and filename are added programmatically. If an error occurs during an attempt to open any file, the client is redirected to **error.html**.

Line 51 instantiates a DOM parser and assigns it to variable **reader**. Line 52 loads and parses **forums.xml**; the **Document** object created is assigned to variable **document**. Because we wish to create a **forum** element within **forums**, line 55 calls the **Document** object's **createElement** method with the name of the new element (**"forum"**). The **filename** attribute of the new **Element** node is set by calling ***setAttribute*** and passing the attribute's name and value.

The **forum** element contains only one piece of information—the forum name—added by lines 58–61. Line 58 creates another **Element** node named **name**. To add character data to the new **Element** node, a child **Text** node must be created. We call method **createTextNode** (line 59) with the forum name from the form (i.e., **form["name"].value**). Line 60 appends the **Text** node to the **Element** node referenced by **name** by calling method **appendChild**. Line 61 adds the **Element** node referenced by **name** to the **Element** node referenced by **forum**.

Line 64 accesses the **documentElement** attribute of **document** to obtain the root element node (i.e., **forums**). Lines 65–66 obtain a **NodeList** of all **forum** elements by calling method **getElementsByTagName**, the first of which is assigned to variable **firstForum**. Line 67 inserts the new **Element** node referenced by **forum** before the first child node of **forums** by calling method **insertBefore**. With this technique, the most recently added forums appear first in the forum list.

To update **forums.xml**, line 70 **seek**s to the beginning and deletes any existing data (by truncating the file to size **0**). Line 72 then calls function **PrettyPrint** to write the updated XML to **forumsFile**.

Line 76 loads and parses file **template.xml** (Fig. 16.25) by calling method **fromStream** and assigns the **Document** object created to variable **document**. Line 77 uses **documentElement** to get the root element, and line 78 sets its **file** attribute's value to the specified filename. Lines 81–84 add the **name** node, and lines 87–88 output the updated XML to **newForumFile** and close the file. Lines 89–90 close **template.xml** and release the **Document** object from memory. The user is redirected to **default.py** in line 92.

Figure 16.26 contains the Python script that allows users to add messages to a forum. When **formatting.xsl** (Fig. 16.27) is applied to a forum document, a link to **addPost.py** is added to the page, which includes the current forum's filename. This filename is passed to **addPost.py** (e.g., **addPost.py?file=forum1.xml**).

```
<?xml version = "1.0"?>

<!-- Fig. 16.25: template.xml -->
<!-- Empty forum file         -->

<?xml:stylesheet type = "text/xsl" href = "../XML/formatting.xsl"?>
<forum>
</forum>
```

Fig. 16.25 XML template for generating new forums.

```
#!c:\Python\python.exe
# Fig. 16.26: addPost.py
# Adds a message to a forum.

import re
import os
import sys
import cgi
import time

# 4DOM packages
from xml.dom.ext.reader import PyExpat
from xml.dom.ext import PrettyPrint

def printHeader( title, style ):
   print """Content-type: text/html

<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml">

<head>
<title>%s</title>
<link rel = "stylesheet" href = "%s" type = "text/css" />
</head>

<body>""" % ( title, style )

# identify client browser
if os.environ[ "HTTP_USER_AGENT" ].find( "MSIE" ) != -1:
   prefix = "../XML/"    # Internet Explorer
else:
   prefix = "forum.py?file="

form = cgi.FieldStorage()

# user has submitted message to post
if form.has_key( "submit" ):
   filename = form[ "file" ].value

   # add message to forum
   if not re.match( "\w+\.xml$", filename  ):
      print "Location: /error.html\n"
      sys.exit()

   try:
      forumFile = open( "../htdocs/XML/" + filename, "r+" )
   except IOError:
      print "Location: /error.html\n"
      sys.exit()

```

Fig. 16.26 Script that adds a message to a forum. (Part 1 of 3.)

```
   # parse forum document
   reader = PyExpat.Reader()
   document = reader.fromStream( forumFile )
   documentNode = document.documentElement

   # create message element
   message = document.createElement( "message" )
   message.setAttribute( "timestamp", time.ctime( time.time() ))

   # add elements to message
   messageElements = [ "user", "title", "text" ]

   for item in messageElements:

      if not form.has_key( item ):
         text = "( Field left blank )"
      else:
         text = form[ item ].value

      # create nodes
      element = document.createElement( item )
      elementText = document.createTextNode( text )
      element.appendChild( elementText )
      message.appendChild( element )

   # append new message to forum and update document on disk
   documentNode.appendChild( message )
   forumFile.seek( 0, 0 )
   forumFile.truncate()
   PrettyPrint( document, forumFile )
   forumFile.close()
   reader.releaseNode( document )

   print "Location: %s\n" % ( prefix + form[ "file" ].value )

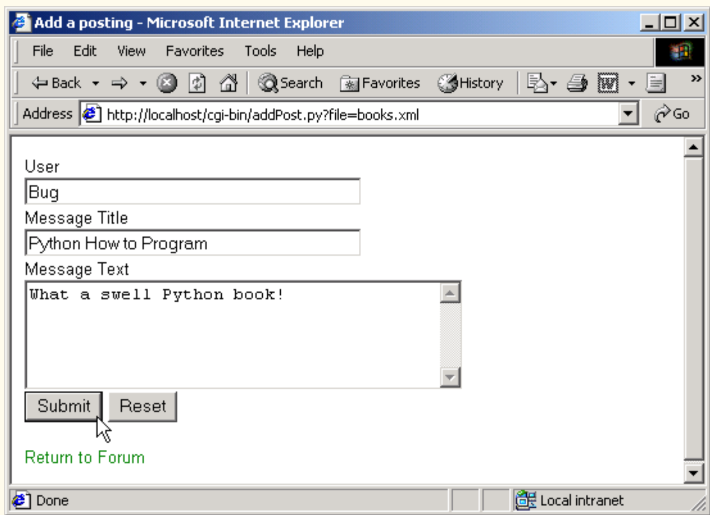
# create form to obtain new message
elif form.has_key( "file" ):
   printHeader( "Add a posting", "/XML/site.css" )
   print """\n<form action = "addPost.py" method="post">
User<br />
<input type = "text" name = "user" size = "40" /><br />
Message Title<br />
<input type = "text" name = "title" size = "40" /><br />
Message Text<br />
<textarea name = "text" cols = "40" rows = "5"></textarea><br />
<input type = "hidden" name = "file" value = "%s" />
<input type = "submit" name = "submit" value = "Submit" />
<input type = "reset" value = "Reset" />
</form>

<a href = "%s">Return to Forum</a>
</body>

```

Fig. 16.26 Script that adds a message to a forum. (Part 2 of 3.)

```
107 </html>""" % ( form[ "file" ].value,
108     prefix + form[ "file" ].value )
109 else:
110     print "Location: /error.html\n"
```



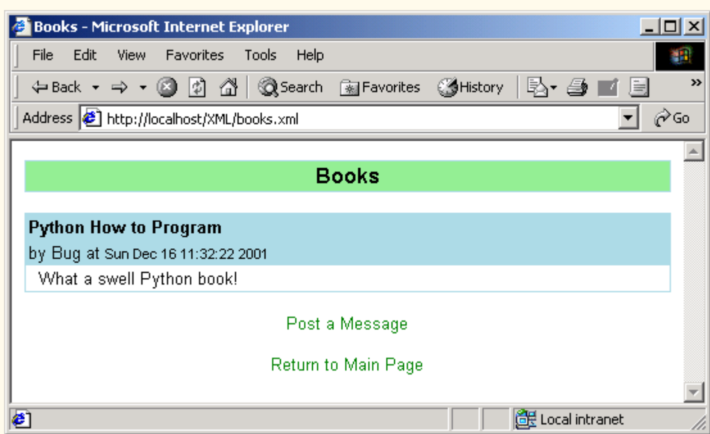


Fig. 16.26 Script that adds a message to a forum. (Part 3 of 3.)

Line 37 obtains the form values posted to the script. The user has not yet submitted a new message; therefore, the form does not contain the value **"submit"** (line 40), and execution proceeds to line 90. If the form contains a single value (i.e., the filename), lines 91–108 output a form, which includes fields for the user name, message title, message text and the forum filename as a hidden value (line 99). Note that, if no parameters are passed to the script, the script has been accessed in an inappropriate way, and the programs redirects the browser to **error.html** (line 110).

When the form data are submitted, the posted information is processed, starting at line 41. As in the previous figure, the filename is checked for an **.xml** extension, and the file

is opened (lines 44–52). Lines 55–61 parse the forum file, create an **Element** node with tag name **message** and set the node's **timestamp** attribute by calling method **setAttribute**.

Lines 64–77 create **Element** nodes that represent the **user**, **title** and **text** and add text that corresponds to the values entered in the form. Note that, if a field has been left blank, **"(Field left blank)"** is entered for that field. Each new **Element** node is appended to the node referenced by **message** (line 77).

Line 80 appends the node referenced by **message** to the node referenced by **forum**. Lines 81–82 then **seek** and **truncate** the XML file to eliminate the file's content and write the updated XML markup. Lines 84–85 close the file and free the **Document** object from memory. The user is redirected to the updated XML document in line 87.

16.6.3 Alterations for Browsers without XML and XSLT Support

This case study uses an XSLT style sheet (**formatting.xsl** in Fig. 16.27) to transform XML data into XHTML that is rendered in Internet Explorer. Recall that each XML document sent to Internet Explorer contains a processing instruction that references this style sheet.

Support for XSLT currently is available only for Internet Explorer 5 and higher. This means that our message forum application could send XML content to some browsers (e.g., Netscape Communicator 6) that do not have built-in XML parsers and XSLT processors. To create a more client-independent application, we can parse the XML on the server and apply the XSLT transformation on the server.

```
<?xml version = "1.0"?>

<!-- Fig. 16.27: formatting.xsl  -->
<!-- Style sheet for forum files -->

<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <!-- match document root -->
   <xsl:template match = "/">
      <html xmlns = "http://www.w3.org/1999/xhtml">

      <!-- apply templates for all elements -->
      <xsl:apply-templates select = "*" />
      </html>
   </xsl:template>

   <!-- match forum elements -->
   <xsl:template match = "forum">
      <head>
         <title><xsl:value-of select = "name" /></title>
         <link rel = "stylesheet" type = "text/css"
            href = "../XML/site.css" />
      </head>

```

Fig. 16.27 XSLT style sheet that transforms XML into XHTML. (Part 1 of 3.)

```
      <body>
         <table width = "100%" cellspacing = "0"
            cellpadding = "2">
            <tr>
               <td class = "forumTitle">
                  <xsl:value-of select = "name" />
               </td>
            </tr>
         </table>

         <!-- apply templates for message elements -->
         <br />
            <xsl:apply-templates select = "message" />
         <br />

         <div style = "text-align: center">
            <a>

               <!-- add href attribute to "a" element -->
               <xsl:attribute name = "href">../cgi-bin/
addPost.py?file=<xsl:value-of select = "@file" />
               </xsl:attribute>
               Post a Message
            </a>
            <br /><br />
          <a href = "../cgi-bin/default.py">Return to Main Page</a>
         </div>

      </body>
   </xsl:template>

   <!-- match message elements -->
   <xsl:template match = "message">
      <table width = "100%" cellspacing = "0"
         cellpadding = "2">
         <tr>
            <td class = "msgTitle">
               <xsl:value-of select = "title" />
            </td>
         </tr>

         <tr>
            <td class = "msgInfo">
               by
               <xsl:value-of select = "user" />
               at
               <span class = "date">
                  <xsl:value-of select = "@timestamp" />
               </span>
            </td>
         </tr>

```

Fig. 16.27 XSLT style sheet that transforms XML into XHTML. (Part 2 of 3.)

```
            <tr>
               <td class = "msgText">
                  <xsl:value-of select = "text" />
               </td>
            </tr>

         </table>
      </xsl:template>

   </xsl:stylesheet>
```

Fig. 16.27 XSLT style sheet that transforms XML into XHTML. (Part 3 of 3.)

4XSLT, which is a package included in **4Suite**, contains an XSLT processor for transforming XML into HTML. We can create an instance of this processor for applying style sheets to XML documents.

Recall how the **prefix** variable in **default.py** (Fig. 16.23) and **addPost.py** (Fig. 16.26) was used to define where links or redirection statements sent clients. By allowing Internet Explorer's XML parser and XSLT processor to parse the XML and apply a style sheet to the XML, we reduce the load on the server. For browsers without XML and XSLT support, however, we direct clients to a Python script that parses the XML document and sends HTML to the client.

We therefore insert a browser test at line 44 of **default.py** and at line 32 of **addPost.py**:

```
if os.environ[ "HTTP_USER_AGENT" ].find( "MSIE" ) != -1:
   prefix = "../XML/"
else:
   prefix = "forum.py?file="
```

Variable **prefix** is set according to whether ***MSIE*** (Microsoft Internet Explorer) appears in the ***HTTP_USER_AGENT*** environment variable. For simplicity, we assume Internet Explorer 5 or higher (with msxml 3.0 or higher) is the only version of MSIE being used and do not test for older versions.

Once **prefix** has been set, we may use its value to customize the URLs generated by the scripts. One example occurs in line 87 of **addPost.py**:

```
print "Location: %s\n" % ( prefix + form[ "file" ].value )
```

This line directs Internet Explorer users to the specified XML forum file located in **../XML/**, but sends users of other browsers to **forum.py**, a Python script that receives a single parameter (i.e., the filename).

Figure 16.28 shows **forum.py**, which transforms XML documents to HTML on the server. The figure also includes the rendered HTML output displayed in Netscape Communicator.

If a filename is not passed to the script, the user is redirected to **error.html** (line 40). Otherwise, execution begins at line 16. Lines 16–18 determine whether the specified filename ends in **.xml**. If so, lines 21–22 open the XSLT style sheet (**formatting.xsl**) and the specified XML document, respectively. If an error occurs during an attempt to open one of these files, the user is redirected to **error.html** (line 24).

The XML then is transformed into HTML for display. Line 28 instantiates a **4XSLT** ***Processor*** object, which transforms XML into HTML, by applying an XSLT style sheet. Line 31 specifies the appropriate XSLT style sheet by invoking **processor**'s ***appendStyleSheetStream*** method. This method appends a style sheet to the list of style sheets a **Processor** can use. Note that more than one style sheet can be appended (i.e., **appendStyleSheetStream** can be called multiple times) so that the same **Processor** object can be used to transform an XML document to many different formats. The argument passed to **appendStyleSheetStream** must be a Python file object. Other methods for appending style sheets to a **4XSLT Processor** are ***appendStyleSheetString***, ***appendStyleSheetNode*** and ***appendStyleSheetUri***, which accept as arguments a string containing an XSLT style sheet, a DOM tree containing a style sheet and a URI that references a style sheet, respectively. The specified URI may be a URL (in the form of a string) that represents the location of the style sheet on the Web.

```
1  #!c:\Python\python.exe
2  # Fig. 16.28: forum.py
3  # Display forum postings for non-Internet Explorer browsers.
4
5  import re
6  import cgi
7  import sys
8  from xml.xslt import Processor
9
10 form = cgi.FieldStorage()
11
12 # form to display has been specified
13 if form.has_key( "file" ):
14
15    # determine whether file is xml
16    if not re.match( "\w+\.xml$", form[ "file" ].value ):
17       print "Location: /error.html\n"
18       sys.exit()
19
20    try:
21       style = open( "../htdocs/XML/formatting.xsl" )
22       XMLFile = open( "../htdocs/XML/" + form[ "file" ].value )
23    except IOError:
24       print "Location: /error.html\n"
25       sys.exit()
26
27    # create XSLT processor instance
28    processor = Processor.Processor()
29
30    # specify style sheet
31    processor.appendStylesheetStream( style )
32
```

Fig. 16.28 Script that transforms XML into HTML for browsers without XSLT support. (Part 1 of 2.)

```
33      # apply style sheet to XML document
34      results = processor.runStream( XMLFile )
35      style.close()
36      XMLFile.close()
37      print "Content-type: text/html\n"
38      print results
39   else:
40      print "Location: /error.html\n"
```

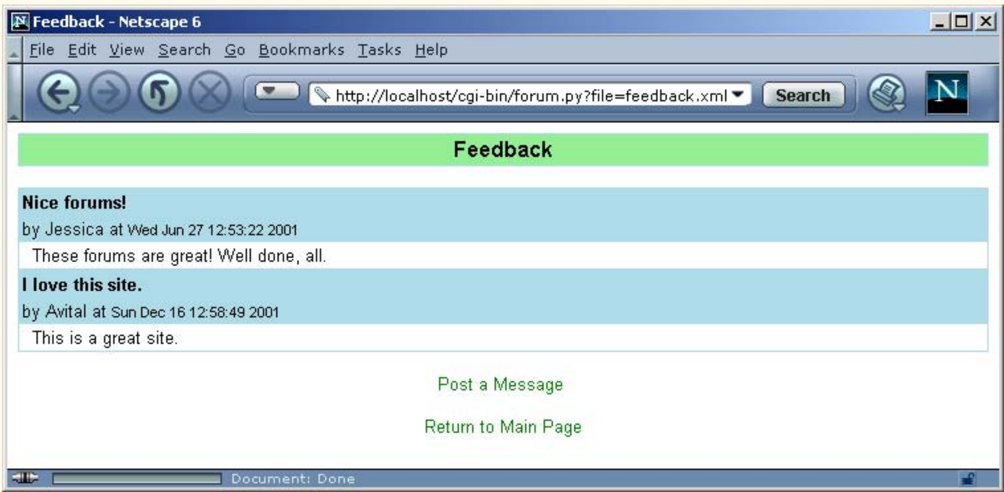


Fig. 16.28 Script that transforms XML into HTML for browsers without XSLT support. (Part 2 of 2.)

Line 34 invokes the **`Processor`**'s ***`runStream`*** method to apply the style sheet to the XML document. As with **`appendStyleSheetStream`**, the object passed to **`runStream`** must be a Python file object. Other methods used for applying style sheets are ***`runString`***, ***`runNode`*** and ***`runUri`***, which accept as arguments a string containing XML, a DOM tree containing XML and a URI that references an XML document, respectively.

Lines 35–36 close the XSLT and XML files used by the script. Line 37 prints the content type header for the Web browser. The transformed XML is then sent to the client as HTML (line 38).

In this chapter, we used the concepts presented in Chapter 15 to create XML-based applications. We used Python packages containing DOM implementations and SAX implementations to parse our XML documents, then used XSLT style sheets to display the XML document content in a browser. In Chapter 17, Database Application Programming Interface (DB-API), we discuss databases, the widely employed relational database model and the Structured Query Language (SQL), a language used to obtain database contents easily.

16.7 Internet and World Wide Web Resources

pyxml.sourceforge.net
The home page for **PyXML**, a Python XML processing package. **PyXML** contains several tools, such as a DOM-based and a SAX-based validating XML parsers.

4suite.org
The home page for **4Suite**, a Python XML processing package. **4Suite** contains several DOM implementations for DOM-based parsing and tools for other XML-related technologies.

www.python.org/doc/current/lib/content-handler-objects.html
This site contains documentation for **xml.sax.ContentHandler** event handlers.

SUMMARY

- Support for XML is provided through a large collection of freely available packages.
- The process by which Python applications can generate XML dynamically is similar to that by which they generate XHTML. For example, to output XML from a Python script, we can use **print** statements or we can use XSLT.
- The modules included with Python for DOM manipulation are **xml.minidom** and **xml.pulldom**. However, neither of these DOM implementations is fully compliant with the W3C's DOM Recommendation.
- A third-party package called **4DOM** is a fully compliant DOM implementation. **4DOM** is included with XML package **PyXML** (**pyxml.sourceforge.net**). Once **PyXML** is installed, the extended DOM components of **4DOM** are accessed via **xml.dom.ext**.
- **4XSLT**, used for applying a style sheet to an XML document, is located in another XML package called **4Suite** (**4suite.org**), from Fourthought, Inc.
- **4DOM**'s **reader** package includes module **PyExpat**.
- **PyExpat** contains class **Reader**, an XML parser. A **Reader** object takes an XML document and parses it, storing it in memory as a tree structure (called a DOM tree).
- The **Node** class, or a class derived from **Node**, represents an XML element, node, comment, etc. in an XML document. Other classes include **NodeList**, an ordered list of nodes, and **NamedNodeMap**, a dictionary of attribute nodes.
- A **Document** object represents the entire XML document (in memory) and provides methods for manipulating its data.
- **Element** nodes represent XML elements.
- **Text** nodes represent character data.
- **Attr** nodes represent attributes in start tags.
- **Comment** nodes represent comments.
- **Document** nodes can contain **Element**, **Text** and **Comment** nodes.
- **Element** nodes can contain **Attr**, **Element**, **Text** and **Comment** nodes.
- Method **fromStream** accepts as input a Python file object and returns a **Document** object.
- A **Document** object's **documentElement** attribute returns the **Document**'s root element node.
- Function **StripXml** removes insignificant whitespace from an XML DOM tree.
- A **Node** object's **childNodes** attribute contains a list of that **Node**'s children.
- A **Node** object's **firstChild** attribute corresponds to the first child in that **Node**'s list of children.

- Parent nodes are obtained through the **parentNode** attribute.
- Method **releaseNode** removes a specified **Document** (i.e., DOM tree) from memory.
- Method **getElementsByTagName** returns a **NodeList** whose **Element** nodes have a particular tag name.
- A **Document** object's **createElement** method creates an **Element** node.
- Function **PrettyPrint** writes an XML DOM tree to a specified output stream.
- For SAX-based parsing, programmers use a package that is included with Python (versions 2.0 and higher)—**xml.sax.** Package **xml.sax** contains SAX classes and functions. With SAX-based parsing, the parser reads the input to identify the XML markup. As the parser encounters markup, event handlers (i.e., methods) are called.
- Class **ContentHandler** contains methods for handling SAX events. These methods can be overridden to perform the desired parsing.
- The **xml.sax** function **parse** creates a SAX parser. The document passed to function **parse** may be specified as either a Python file object or a filename. The second argument passed to **parse** must be an instance of class **xml.sax.ContentHandler** (or a derived class of **ContentHandler**), the main callback handler in **xml.sax**.
- If an error occurs during parsing, **parse** raises a **SAXParseException** exception.
- Methods **startElement** and **endElement** are called when element start tags and end tags are encountered, respectively. Method **startElement** takes two arguments—the element's name as a string and the element's attributes. The attributes are passed as an instance of class **AttributesImpl**, defined in **xml.sax.reader**. Method **endElement** executes when an element's end tag is encountered and takes the end tag's name as an argument.
- A **4XSLT Processor** transforms XML to HTML by applying an XSLT style sheet.
- The **Processor**'s **appendStyleSheetStream** method specifies the XSLT style sheet to apply. This method appends a style sheet to the list of style sheets a **Processor** can use. The argument passed to **appendStyleSheetStream** must be a Python file object.
- The **Processor**'s **runStream** method applies the style sheet to the XML document. The object passed to **runStream** must be a Python file object.

TERMINOLOGY

4DOM package
4Suite package
4XSLT package
appendChild method of class **Node**
appendStyleSheetNode method of class **Processor**
appendStyleSheetStream method of class **Processor**
appendStyleSheetString method of class **Processor**
appendStyleSheetUri method of class **Processor**
Attr class
attributes attribute of class **Node**
characters method of class **ContentHandler**
childNodes attribute of class **Node**
Comment class
ContentHandler class
createAttribute method of class **Document**
createComment method of class **Document**
createElement method of class **Document**
createTextNode method of class **Document**
data attribute of class **Comment**
data attribute of class **Text**
Document class
Document Object Model (DOM)
DOM parser
DOM tree
documentElement attribute

Element class
endDocument method of class
  ContentHandler
endElement method of class
  ContentHandler
event handler
firstChild attribute of class **Node**
fromStream method of class **Reader**
getAttribute method of class **Element**
getAttributeNode method of class
  Element
getElementsByTagName method of class
  Document
getElementsByTagName method of class
  Element
insertBefore method of class **Node**
isSameNode method of class **Node**
item method of class **NamedNodeMap**
item method of class **NodeList**
lastChild attribute of class **Node**
length attribute of class **NamedNodeMap**
length attribute of class **NodeList**
name attribute of class **Attr**
NamedNodeMap class
nextSibling attribute of class **Node**
Node class
NodeList class
nodeName attribute of class **Node**
nodeType attribute of class **Node**
nodeValue attribute of class **Node**
parent node
parentNode attribute of class **Node**
parse function of package **xml.sax**
parser
prefix attribute of class **Attr**
PrettyPrint function of package **4DOM**
previousSibling attribute of class **Node**
Processor class
PyExpat module
PyXML package
Reader class
removeAttribute method of class
  Element
removeAttributeNode method of class
  Element
removeChild method of class **Node**
replaceChild method of class **Node**
runNode method
runStream method
runString method
runUri method
SAX-based parsing
SAX parser
setAttribute method of class **Element**
setAttributeNode method of class
  Element
sibling
startDocument method of class
  ContentHandler
startElement method of class
  ContentHandler
StripXml function of package **4DOM**
tagName attribute of class **Element**
Text class

SELF-REVIEW EXERCISES

16.1 Fill in the blanks for each of the following statements:

a) A **PyExpat** __________ object takes an XML document and parses it, storing it in memory as a tree structure.
b) A **Document** object's __________ attribute refers to the **Document**'s root element.
c) **4DOM**'s __________ function prints an XML DOM tree to a specified output stream.
d) **Node** method __________ appends a new child to the list of child nodes.
e) Method __________ removes a specified DOM tree from memory, freeing resources.
f) **xml.sax** class __________ contains methods for handling SAX events which can be overridden to perform desired parsing.
g) A **4XSLT** __________ object transforms XML into HTML, by applying a specified XSLT style sheet.
h) Method **fromStream** returns a __________ object.

16.2 State which of the following statements are *true* and which are *false*. If *false*, explain why.

a) To create a Python script which outputs XML, programmers use module **xmlgen**.
b) Method **insertBefore(** *a*, *b* **)** inserts node *a* before node *b*.
c) The different XML node types are represented in a DOM tree by class **XMLNode**.

d) **Node** attribute **childNodes** returns a **NodeList** object containing the node's children.
e) **4DOM**'s **StripXml** function parses an XML document.
f) With SAX-based parsing, the parser reads the input, storing it in memory as a tree structure.
g) The second argument passed to **parse** must be an instance of class **xml.sax.ContentHandler** (or a subclass of **ContentHandler**).
h) If an error occurs while parsing a file, **parse** raises a **SAXParseException** exception.

ANSWERS TO SELF-REVIEW EXERCISES

16.1 a) **Reader**. b) **documentElement**. c) **PrettyPrint**. d) **appendChild**. e) **releaseNode**. f) **ContentHandler**. g) **Processor**. h) **Document**.

16.2 a) False. Programmers can use **print** statements, XSLT or the DOM to generate XML markup. b) True. c) False. XML node types are represented by classes derived from **Node**. d) True. e) False. **StripXml** removes insignificant whitespace from an XML DOM tree. f) False. With SAX-based parsing, data is not stored in memory. As the parser encounters markup, event handlers are called. g) True. h) True.

EXERCISES

16.3 Modify the program in Fig. 16.13. Allow the user to add a new element to each **contact** element. For instance, if the user adds a **phoneNumber** element, the user should be prompted to provide a phone number for each contact. Each time a user adds a contact, the user should be prompted to provide information for any new elements in addition to the first and last names. Function **printList** should print any new information as well as the contact's first and last names.

16.4 Create a Python script that, given an XML document, creates an XHTML list of the document's elements in hierarchical order. Display the elements in Internet Explorer. For example, given the XML document in Fig. 16.29, create a Python script that lists the elements as shown in Fig. 16.29.

16.5 These lines of code are from lines 45–46 of **formatting.xsl** (Fig. 16.27). Explain why the **@** in front of **"@file"** is necessary in the **xsl:value-of** element.

```
<xsl:attribute name = "href">../cgi-bin/
addPost.py?file=<xsl:value-of select = "@file" />
</xsl:attribute>
```

16.6 Describe the purpose of Fig. 16.27 (**formatting.xsl**).

16.7 Implement the **Delete a Forum** option in **default.py**. Selecting this option should direct the user to a script named **deleteForum.py**. Here, the user can select a forum name from a list. Your script should remove the selected forum from **forums.xml** and delete the underlying XML document. After removing the forum, the script should redirect the browser to **default.py**.

```
<?xml version = "1.0"?>
<?xml:stylesheet type = "text/xsl" href = "games.xsl"?>

<!-- Fig. 16.29                    -->
<!-- Sports Database: sports.xml -->
```

Fig. 16.29 **sports.xml** for Exercise 16.4. (Part 1 of 2.)

```

<sports>
   <game id = "783">
      <name>Cricket</name>
      <summary>
         <paragraph>
            More popular among commonwealth nations.
         </paragraph>
      </summary>
   </game>

   <game id = "239">
      <name>Baseball</name>
      <summary>
         <paragraph>
            More popular in America.
         </paragraph>
      </summary>
   </game>
</sports>
```

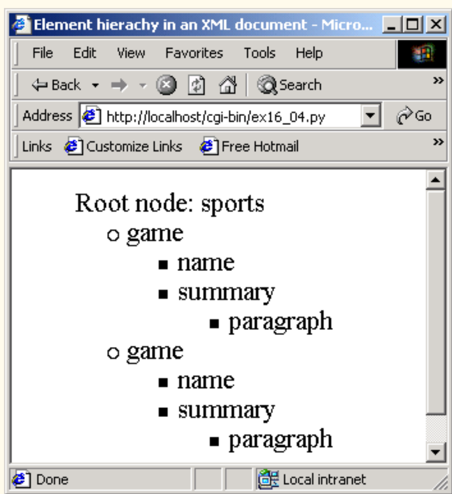


Fig. 16.29 `sports.xml` for Exercise 16.4. (Part 2 of 2.)

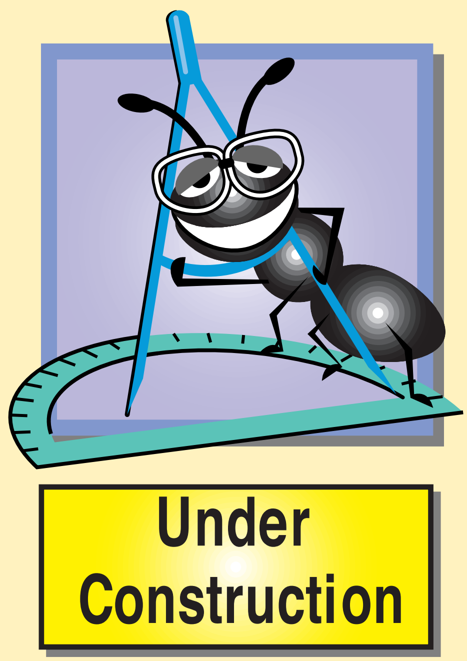
16.8 Implement the **Modify a Forum** option in **`default.py`** such that individual messages can be deleted. Selecting this option should direct the user to a script named **`modifyForum.py`**. Here, the user can select a forum name from a list. Script **`modifyForum.py`** should then display all the messages in the specified forum, allowing the user to select one for deletion. Once selected, **`modifyForum.py`** should remove the given message from the current forum and redirect the browser to **`default.py`**.

17

Database Application Programming Interface (DB-API)

Objectives

- To understand the relational database model.
- To understand basic database queries using Structured Query Language (SQL).
- To use the methods of the `MySQLdb` module to query a database, insert data into a database and update data in a database.



It is a capital mistake to theorize before one has data.
Arthur Conan Doyle

Now go, write it before them in a table, and note it in a book, that it may be for the time to come for ever and ever.
The Holy Bible: The Old Testament

Let's look at the record.
Alfred Emanuel Smith

True art selects and paraphrases, but seldom gives a verbatim translation.
Thomas Bailey Aldrich

Get your facts first, and then you can distort them as much as you please.
Mark Twain

I like two kinds of men: domestic and foreign.
Mae West

Outline

17.1 Introduction

In Chapter 14, File Processing and Serialization, we discussed sequential-access and random-access file processing. Sequential-file processing is appropriate for applications in which most or all of the file's information is to be processed. On the other hand, random-access file processing is appropriate for applications in which only a small portion of a file's data is to be processed. For instance, in transaction processing it is crucial to locate and, possibly, update an individual piece of data quickly. Python provides capabilities for both types of file processing.

A *database* is an integrated collection of data. Many companies maintain databases to organize employee information, such as names, addresses and phone numbers. There are many different strategies for organizing data to facilitate easy access and manipulation of the data. A *database management system (DBMS)* provides mechanisms for storing and organizing data in a manner consistent with the database's format. Database management systems allow for the access and storage of data without concern for the internal representation of databases.

Today's most popular database systems are *relational databases*, which store data in tables and define relationships between the tables. A language called *Structured Query*

Language (*SQL*—pronounced as its individual letters or as “sequel”) is used almost universally with relational database systems to perform *queries* (i.e., to request information that satisfies given criteria) and to manipulate data. [*Note*: The writing in this chapter assumes that SQL is pronounced as its individual letters. For this reason, we often precede SQL with the article “an” as in “an SQL database” or “an SQL statement.”]

Some popular relational database systems include Microsoft SQL Server, Oracle, Sybase, DB2, Informix and MySQL. In this chapter, we present examples using MySQL. All examples in this chapter use MySQL version 3.23.41. [*Note*: The Deitel & Associates, Inc. Web site (**`www.deitel.com`**) provides step-by-step instructions for installing MySQL and helpful MySQL commands for creating, populating and deleting tables.]

A programming language connects to, and interacts with, relational databases via an *interface*—software that facilitates communications between a database management system and a program. Python programmers communicate with databases using modules that conform to the Python *Database Application Programming Interface (DB-API)*. Section 17.5 discusses the DB-API specification.

17.2 Relational Database Model

The *relational database model* is a logical representation of data that allows relationships among data to be considered without concern for the physical structure of the data. A relational database is composed of *tables*. Figure 17.1 illustrates a table that might be used in a personnel system. The table name is **`Employee`**, and its primary purpose is to maintain the specific attributes of various employees. A particular row of the table is called a *record* (or *row*). This table consists of six records. The **`number`** *field* (or *column*) of each record in the table is the *primary key* for referencing data in the table. A primary key is a field (or combination of fields) in a table that contain(s) unique data—i.e, data that is not duplicated in other records of that table. This guarantees that each record can be identified by at least one distinct value. Examples of primary-key fields are columns that contain social security numbers, employee IDs and part numbers in an inventory system. The records of Fig. 17.1 are *ordered* by primary key. In this case, the records are listed in increasing order (they also could be listed in decreasing order).

`Number`	`Name`	`Department`	`Salary`	`Location`
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

Record/Row

Primary key

Field/Column

Fig. 17.1 Relational-database structure of an **`Employee`** table.

Each column of the table represents a different field. Records normally are unique (by primary key) within a table, but particular field values might be duplicated in multiple records. For example, three different records in the **Employee** table's **Department** field contain the number 413.

Often, different users of a database are interested in different data and different relationships among those data. Some users require only subsets of the table columns. To obtain table subsets, we use SQL statements to specify certain data we wish to *select* from a table. SQL provides a complete set of commands (including ***SELECT***) that enable programmers to define complex *queries* to select data from a table. The results of queries commonly are called *result sets* (or *record sets*). For example, we might select data from the table in Fig. 17.1 to create a new result set that contains only the location of each department. This result set appears in Fig. 17.2. SQL queries are discussed in detail in Section 17.4.

17.3 Relational Database Overview: Books Database

This section gives an overview of SQL in the context of a sample **Books** database we created for this chapter. Before we discuss SQL, we overview the tables of the **Books** database. [*Note*: The CD that accompanies this book contains a program called **DBSetup.py** that creates and populates a **Books** database with sample data.]

We use the **Books** database to introduce various database concepts, such as using SQL to obtain useful information from the database and to manipulate the database. We provide the database in the examples directory for this chapter on the CD that accompanies this book. Note that when using MySQL on Windows, the database name is case insensitive (i.e., the **Books** database and the **books** database refer to the same database). However, on Linux, the database name is case sensitive (i.e., the **Books** database and the **books** database refer to different databases).

The database consists of four tables: **Authors**, **Publishers**, **AuthorISBN** and **Titles**. The **Authors** table (described in Fig. 17.3) consists of three fields (or columns) that maintain each author's unique ID number, first name and last name. Figure 17.4 contains the sample data from the **Authors** table of the **Books** database.

The **Publishers** table (described in Fig. 17.5) consists of two fields, which represent each publisher's unique ID and name. Figure 17.6 contains the data from the **Publishers** table of the **Books** database.

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

Fig. 17.2 Result set formed by selecting **Department** and **Location** data from the **Employee** table.

Field	Description
AuthorID	Author's ID number in the database. In the **Books** database, this **int** field is defined as an *auto-incremented field*. For each new record inserted in this table, the database increments the **AuthorID** value, ensuring that each record has a unique **AuthorID**. This field is the table's primary key.
FirstName	Author's first name (a **string**).
LastName	Author's last name (a **string**).

Fig. 17.3 **Authors** table from **Books**.

AuthorID	**FirstName**	**LastName**
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

Fig. 17.4 Data from the **Authors** table of **Books**.

Field	Description
PublisherID	The publisher's ID number in the database. This auto-incremented **int** field is the table's primary-key field.
PublisherName	The name of the publisher (a **string**).

Fig. 17.5 **Publishers** table from **Books**.

The **AuthorISBN** table (described in Fig. 17.7) consists of two fields that maintain the authors' ID numbers and the corresponding ISBN numbers of their books. This table helps associate the names of the authors with the titles of their books. Figure 17.8 contains a portion of the sample data from the **AuthorISBN** table of the **Books** database.

ISBN is an abbreviation for "International Standard Book Number"—a numbering scheme by which publishers worldwide assign every book a unique identification number. [*Note*: To save space, we split the contents of this figure into two columns, each containing the **AuthorID** and **ISBN** fields.]

PublisherID	PublisherName
1	Prentice Hall
2	Prentice Hall PTG

Fig. 17.6 Data from the **Publishers** table of **Books**.

Field	Description
AuthorID	The author's ID number, which allows the database to associate each book with a specific author. The integer ID number in this field must also appear in the **Authors** table.
ISBN	The ISBN number for a book (a **string**).

Fig. 17.7 **AuthorISBN** table from **Books**.

AuthorID	ISBN	AuthorID	ISBN
1	0130895725	1	0130284181
1	0132261197	1	0130895601
1	0130895717	2	0130895725
1	0135289106	2	0132261197
1	0139163050	2	0130895717
1	013028419x	2	0135289106
1	0130161438	2	0139163050
1	0130856118	2	013028419x
1	0130125075	2	0130161438
1	0138993947	2	0130856118
1	0130852473	2	0130125075
1	0130829277	2	0138993947
1	0134569555	2	0130852473
1	0130829293	2	0130829277
1	0130284173	2	0134569555

Fig. 17.8 Data from **AuthorISBN** table in **Books**. [Note: This table shows only a portion of the sample data.]

AuthorID	ISBN	AuthorID	ISBN
2	0130829293	3	0130856118
2	0130284173	3	0134569555
2	0130284181	3	0130829293
2	0130895601	3	0130284173
3	013028419x	3	0130284181
3	0130161438	4	0130895601

Fig. 17.8 Data from **AuthorISBN** table in **Books**. [*Note*: This table shows only a portion of the sample data.]

The **Titles** table (described in Fig. 17.9) consists of seven fields that maintain general information about the books in the database. This information includes each book's ISBN number, title, edition number, copyright year and publisher's ID number, as well as the name of a file that contains an image of the book cover and, finally, each book's price. Figure 17.10 contains the sample data from the **Titles** table.

Field	Description
ISBN	ISBN number of the book (a **string**).
Title	Title of the book (a **string**).
EditionNumber	Edition number of the book (a **string**).
Copyright	Copyright year of the book (an **int**).
PublisherID	Publisher's ID number (an **int**). This value must correspond to an ID number in the **Publishers** table.
ImageFile	Name of the file containing the book's cover image (a **string**).
Price	Suggested retail price of the book (a real number). [*Note*: The prices shown in this database are for example purposes only.]

Fig. 17.9 **Titles** table from **Books**.

ISBN	Title	Edition-Number	Publish-erID	Copy-right	ImageFile	Price
0130923613	Python How to Program	1	1	2002	**python.jpg**	$69.95
0130622214	C# How to Program	1	1	2002	**cshtp.jpg**	$69.95
0130341517	Java How to Program	4	1	2002	**jhtp4.jpg**	$69.95

Fig. 17.10 Data from the **Titles** table of **Books**. (Part 1 of 3.)

ISBN	Title	Edition-Number	Publish-erID	Copy-right	ImageFile	Price
0130649341	The Complete Java Training Course	4	2	2002	`javactc4.jpg`	$109.95
0130895601	Advanced Java 2 Platform How to Program	1	1	2002	`advjhtp1.jpg`	$69.95
0130308978	Internet and World Wide Web How to Program	2	1	2002	`iw3htp2.jpg`	$69.95
0130293636	Visual Basic .NET How to Program	2	1	2002	`vbnet.jpg`	$69.95
0130895636	The Complete C++ Training Course	3	2	2001	`cppctc3.jpg`	$109.95
0130895512	The Complete e-Business & e-Commerce Programming Training Course	1	2	2001	`ebecctc.jpg`	$109.95
013089561X	The Complete Internet & World Wide Web Programming Training Course	2	2	2001	`iw3ctc2.jpg`	$109.95
0130895547	The Complete Perl Training Course	1	2	2001	`perl.jpg`	$109.95
0130895563	The Complete XML Programming Training Course	1	2	2001	`xmlctc.jpg`	$109.95
0130895725	C How to Program	3	1	2001	`chtp3.jpg`	$69.95
0130895717	C++ How to Program	3	1	2001	`cpphtp3.jpg`	$69.95
013028419X	e-Business and e-Commerce How to Program	1	1	2001	`ebechtp1.jpg`	$69.95
0130622265	Wireless Internet and Mobile Business How to Program	1	1	2001	`wireless.jpg`	$69.95
0130284181	Perl How to Program	1	1	2001	`perlhtp1.jpg`	$69.95
0130284173	XML How to Program	1	1	2001	`xmlhtp1.jpg`	$69.95
0130856118	The Complete Internet and World Wide Web Programming Training Course	1	2	2000	`iw3ctc1.jpg`	$109.95

Fig. 17.10 Data from the `Titles` table of `Books`. (Part 2 of 3.)

ISBN	Title	EditionNumber	PublisherID	Copyright	ImageFile	Price
0130125075	Java How to Program (Java 2)	3	1	2000	`jhtp3.jpg`	$69.95
0130852481	The Complete Java 2 Training Course	3	2	2000	`javactc3.jpg`	$109.95
0130323640	e-Business and e-Commerce for Managers	1	1	2000	`ebecm.jpg`	$69.95
0130161438	Internet and World Wide Web How to Program	1	1	2000	`iw3htp1.jpg`	$69.95
0130132497	Getting Started with Visual C++ 6 with an Introduction to MFC	1	1	1999	`gsvc.jpg`	$49.95
0130829293	The Complete Visual Basic 6 Training Course	1	2	1999	`vbctc1.jpg`	$109.95
0134569555	Visual Basic 6 How to Program	1	1	1999	`vbhtp1.jpg`	$69.95
0132719746	Java Multimedia Cyber Classroom	1	2	1998	`javactc.jpg`	$109.95
0136325890	Java How to Program	1	1	1998	`jhtp1.jpg`	$69.95
0139163050	The Complete C++ Training Course	2	2	1998	`cppctc2.jpg`	$109.95
0135289106	C++ How to Program	2	1	1998	`cpphtp2.jpg`	$49.95
0137905696	The Complete Java Training Course	2	2	1998	`javactc2.jpg`	$109.95
0130829277	The Complete Java Training Course (Java 1.1)	2	2	1998	`javactc2.jpg`	$99.95
0138993947	Java How to Program (Java 1.1)	2	1	1998	`jhtp2.jpg`	$49.95
0131173340	C++ How to Program	1	1	1994	`cpphtp1.jpg`	$69.95
0132261197	C How to Program	2	1	1994	`chtp2.jpg`	$49.95
0131180436	C How to Program	1	1	1992	`chtp.jpg`	$69.95

Fig. 17.10 Data from the **`Titles`** table of **`Books`**. (Part 3 of 3.)

Figure 17.11 illustrates the relationships among the tables in the **`Books`** database. The first line in each table is the table's name. The field whose name appears in italics contains that table's primary key. A table's primary key uniquely identifies each record in the table. Every record must have a value in the primary-key field, and the value must be unique. This

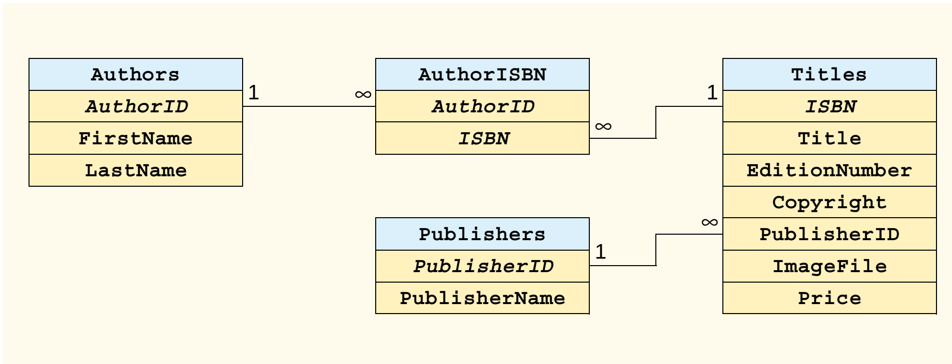


Fig. 17.11 Table relationships in **Books**.

is known as the *Rule of Entity Integrity*. Note that the **AuthorISBN** table contains two fields whose names are italicized. This indicates that these two fields form a *compound primary key*—each record in the table must have a unique **AuthorID**–**ISBN** combination. For example, several records might have an **AuthorID** of **2**, and several records might have an **ISBN** of **0130895601**, but only one record can have both an **AuthorID** of **2** and an **ISBN** of **0130895601**.

Common Programming Error 17.1

Failure to provide a value for a primary-key field in every record breaks the Rule of Entity Integrity and causes the DBMS to report an error.

Common Programming Error 17.2

Providing duplicate values for the primary-key field of multiple records causes the DBMS to report an error.

The lines connecting the tables in Fig. 17.11 represent the *relationships* among the tables. Consider the line between the **Publishers** and **Titles** tables. On the **Publishers** end of the line, there is a **1**, and, on the **Titles** end, there is an infinity (∞) symbol. This line indicates a *one-to-many relationship*, in which every publisher in the **Publishers** table can have an arbitrarily large number of books in the **Titles** table. Note that the relationship line links the **PublisherID** field in the **Publishers** table to the **PublisherID** field in **Titles** table. In the **Titles** table, the **PublisherID** field is a *foreign key*—a field for which every entry has a unique value in another table and where the field in the other table is the primary key for that table (e.g., **PublisherID** in the **Publishers** table). Programmers specify foreign keys when creating a table. The foreign key helps maintain the *Rule of Referential Integrity*: Every foreign-key field value must appear in another table's primary-key field. Foreign keys enable information from multiple tables to be *joined* together for analysis purposes. There is a one-to-many relationship between a primary key and its corresponding foreign key. This means that a foreign-key field value can appear many times in its own table, but must appear exactly once as the primary key of another table. The line between the tables represents the link between the foreign key in one table and the primary key in another table.

Common Programming Error 17.3

Providing a foreign-key value that does not appear as a primary-key value in another table breaks the Rule of Referential Integrity and causes the DBMS to report an error.

The line between the **`AuthorISBN`** and **`Authors`** tables indicates that, for each author in the **`Authors`** table, the **`AuthorISBN`** table can contain an arbitrary number of ISBNs for books written by that author. The **`AuthorID`** field in the **`AuthorISBN`** table is a foreign key of the **`AuthorID`** field (the primary key) of the **`Authors`** table. Note, again, that the line between the tables links the foreign key in table **`AuthorISBN`** to the corresponding primary key in table **`Authors`**. The **`AuthorISBN`** table links information in the **`Titles`** and **`Authors`** tables.

The line between the **`Titles`** and **`AuthorISBN`** tables illustrates another one-to-many relationship; a title can be written by any number of authors. In fact, the sole purpose of the **`AuthorISBN`** table is to represent a many-to-many relationship between the **`Authors`** and **`Titles`** tables; an author can write any number of books, and a book can have any number of authors.

17.4 Structured Query Language (SQL)

This section provides an overview of Structured Query Language (SQL) in the context of our **`Books`** sample database. The SQL queries discussed here form the foundation for the SQL used in the chapter examples.

Figure 17.12 lists SQL keywords and provides a description of each. In the next several subsections, we discuss these SQL keywords in the context of complete SQL queries. Other SQL keywords exist, but are beyond the scope of this text. [*Note*: To locate additional information on SQL, please refer to the bibliography at the end of this chapter.]

SQL keyword	Description
`SELECT`	Selects (retrieves) fields from one or more tables.
`FROM`	Specifies tables from which to get fields or delete records. Required in every **`SELECT`** and **`DELETE`** statement.
`WHERE`	Specifies criteria that determine the rows to be retrieved.
`INNER JOIN`	Joins records from multiple tables to produce a single set of records.
`GROUP BY`	Specifies criteria for grouping records.
`ORDER BY`	Specifies criteria for ordering records.
`INSERT`	Inserts data into a specified table.
`UPDATE`	Updates data in a specified table.
`DELETE`	Deletes data from a specified table.

Fig. 17.12 SQL query keywords.

17.4.1 Basic `SELECT` Query

Let us consider several SQL queries that extract information from database **`Books`**. A typical SQL query "selects" information from one or more tables in a database. Such selections are performed by ***`SELECT`** queries*. The basic format for a **`SELECT`** query is:

```
SELECT * FROM tableName
```

In this query, the asterisk (**`*`**) indicates that all columns from the *tableName* table of the database should be selected. For example, to select the entire contents of the **`Authors`** table (i.e., all data depicted in Fig. 17.4), use the query:

```
SELECT * FROM Authors
```

To select specific fields from a table, replace the asterisk (**`*`**) with a comma-separated list of the field names to select. For example, to select only the fields **`AuthorID`** and **`LastName`** for all rows in the **`Authors`** table, use the query:

```
SELECT AuthorID, LastName FROM Authors
```

This query returns only the data presented in Fig. 17.13. The result set contains the columns in the order that are specified by the query. [*Note*: If a field name contains spaces, the entire field name must be enclosed in square brackets (**`[]`**) in the query. For example, if the field name is **`First Name`**, it must appear in the query as **`[First Name]`**.]

Common Programming Error 17.4

*If a program assumes that an SQL statement using the asterisk (**`*`**) to select fields always returns those fields in the same order, the program could process the result set incorrectly. If the field order in the database table(s) changes, the order of the fields in the result set would change accordingly.*

Performance Tip 17.1

If a program does not know the order of fields in a result set, the program must process the fields by name. This could require a linear search of the field names in the result set. If users specify the field names that they wish to select from a table (or several tables), the application receiving the result set knows the order of the fields in advance. When this occurs, the program can process the data more efficiently, because fields can be accessed directly by column number.

`AuthorID`	`LastName`	`AuthorID`	`LastName`
1	Deitel	8	McPhie
2	Deitel	9	Yaeger
3	Nieto	10	Zlatkina
4	Steinbuhler	12	Wiedermann
5	Santry	12	Liperi
6	Lin	13	Listfield
7	Sadhu		

Fig. 17.13 **`AuthorID`** and **`LastName`** from the **`Authors`** table.

17.4.2 WHERE Clause

In most cases, users search a database for records that satisfy certain *selection criteria*. Only records that match the selection criteria are selected. SQL uses the optional ***WHERE*** *clause* in a **SELECT** query to specify the selection criteria for the query. The simplest format for a **SELECT** query that includes selection criteria is:

```
SELECT fieldName1, fieldName2, … FROM tableName WHERE criteria
```

For example, to select the **Title**, **EditionNumber** and **Copyright** fields from those rows of table **Titles** in which the **Copyright** date is greater than **1999**, use the query:

```
SELECT Title, EditionNumber, Copyright
FROM Titles
WHERE Copyright > 1999
```

Figure 17.14 shows the result set of the preceding query. [*Note*: When we construct a query for use in Python, we create a string containing the entire query. However, when we display queries in the text, we often use multiple lines and indentation to enhance readability.]

Performance Tip 17.2

Using selection criteria improves performance, because queries that involve such criteria normally select a portion of the database that is smaller than the entire database. Working with a smaller portion of the data is more efficient than working with the entire set of data stored in the database.

Title	EditionNumber	Copyright
Internet and World Wide Web How to Program	2	2002
Java How to Program	4	2002
The Complete Java Training Course	4	2002
The Complete e-Business & e-Commerce Programming Training Course	1	2001
The Complete Internet & World Wide Web Programming Training Course	2	2001
The Complete Perl Training Course	1	2001
The Complete XML Programming Training Course	1	2001
C How to Program	3	2001
C++ How to Program	3	2001
The Complete C++ Training Course	3	2001
e-Business and e-Commerce How to Program	1	2001
Internet and World Wide Web How to Program	1	2000
The Complete Internet and World Wide Web Programming Training Course	1	2000
Java How to Program (Java 2)	3	2000

Fig. 17.14 Titles with copyrights after 1999 from table **Titles**. (Part 1 of 2.)

`Title`	`EditionNumber`	`Copyright`
The Complete Java 2 Training Course	3	2000
XML How to Program	1	2001
Perl How to Program	1	2001
Advanced Java 2 Platform How to Program	1	2002
e-Business and e-Commerce for Managers	1	2000
Wireless Internet and Mobile Business How to Program	1	2001
C# How To Program	1	2002
Python How to Program	1	2002
Visual Basic .NET How to Program	2	2002

Fig. 17.14 Titles with copyrights after 1999 from table **`Titles`**. (Part 2 of 2.)

The **`WHERE`** clause condition can contain operators **`<`**, **`>`**, **`<=`**, **`>=`**, **`=`**, **`<>`** and **`LIKE`**. Operator **`LIKE`** is used for *pattern matching* with wildcard characters *percent* (**`%`**) and *underscore mark* (**`_`**). Pattern matching allows SQL to search for strings that "match a pattern."

A pattern that contains a percent (**`%`**) searches for strings in which zero or more characters take the percent character's place in the pattern. For example, the following query locates the records of all authors whose last names start with the letter **`D`**:

```
SELECT AuthorID, FirstName, LastName
FROM Authors
WHERE LastName LIKE 'D%'
```

The preceding query selects the two records shown in Fig. 17.15, because two of the authors in our database have last names that begin with the letter **`D`** (followed by zero or more characters). The **`%`** in the **`WHERE`** clause's **`LIKE`** pattern indicates that any number of characters can appear after the letter **`D`** in the **`LastName`** field. Notice that the pattern string is surrounded by single-quote characters.

Portability Tip 17.1

*Not all database systems support the **`LIKE`** operator, so be sure to read the database system's documentation carefully before employing this operator.*

Portability Tip 17.2

*Some databases use the **`*`** character in place of the **`%`** character in **`LIKE`** expressions.*

`AuthorID`	`FirstName`	`LastName`
1	Harvey	Deitel
2	Paul	Deitel

Fig. 17.15 Authors from the **`Authors`** table whose last names start with **`D`**.

Portability Tip 17.3

In some databases, string data is case sensitive.

Portability Tip 17.4

In some databases, table names and field names are case sensitive.

Good Programming Practice 17.1

By convention, SQL keywords should be written entirely in uppercase letters on systems that are not case sensitive. This emphasizes the SQL keywords in an SQL statement.

A pattern string including an underscore (_) character searches for strings in which exactly one character takes the underscore's place in the pattern. For example, the following query locates the records of all authors whose last names start with any character (specified with _), followed by the letter **i**, followed by any number of additional characters (specified with **%**):

```
SELECT AuthorID, FirstName, LastName
FROM Authors
WHERE LastName LIKE '_i%'
```

The preceding query produces the records listed in Fig. 17.16; five authors in our database have last names in which the letter **i** is the second letter.

Portability Tip 17.5

Some databases use the **?** *character in place of the _ character in* **LIKE** *expressions.*

17.4.3 ORDER BY Clause

The results of a query can be arranged in ascending or descending order using the optional ***ORDER BY*** *clause*. The simplest forms for an **ORDER BY** clause are:

```
SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field ASC
SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field DESC
```

where **ASC** specifies ascending order (lowest to highest), **DESC** specifies descending order (highest to lowest) and *field* specifies the field whose values determine the sorting order.

AuthorID	FirstName	LastName
3	Tem	Nieto
6	Ted	Lin
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

Fig. 17.16 Authors from table **Authors** whose last names contain **i** as the second letter.

For example, to obtain a list of authors arranged in ascending order by last name (Fig. 17.17), use the query:

```
SELECT AuthorID, FirstName, LastName
FROM Authors
ORDER BY LastName ASC
```

Note that the default sorting order is ascending; therefore, **ASC** is optional.

To obtain the same list of authors arranged in descending order by last name (Fig. 17.18), use the query:

```
SELECT AuthorID, FirstName, LastName
FROM Authors
ORDER BY LastName DESC
```

AuthorID	FirstName	LastName
2	Paul	Deitel
1	Harvey	Deitel
6	Ted	Lin
12	Jonathan	Liperi
13	Jeffrey	Listfield
8	David	McPhie
3	Tem	Nieto
7	Praveen	Sadhu
5	Sean	Santry
4	Kate	Steinbuhler
11	Ben	Wiedermann
9	Cheryl	Yaeger
10	Marina	Zlatkina

Fig. 17.17 Authors from table **Authors** in ascending order by **LastName**.

AuthorID	FirstName	LastName
10	Marina	Zlatkina
9	Cheryl	Yaeger
11	Ben	Wiedermann
4	Kate	Steinbuhler
5	Sean	Santry

Fig. 17.18 Authors from table **Authors** in descending order by **LastName**. (Part 1 of 2.)

AuthorID	FirstName	LastName
7	Praveen	Sadhu
3	Tem	Nieto
8	David	McPhie
13	Jeffrey	Listfield
12	Jonathan	Liperi
6	Ted	Lin
2	Paul	Deitel
1	Harvey	Deitel

Fig. 17.18 Authors from table **Authors** in descending order by **LastName**. (Part 2 of 2.)

The **ORDER BY** clause also can be used to order records by multiple fields. Such queries are written in the form:

```
ORDER BY field1 sortingOrder, field2 sortingOrder, …
```

where *sortingOrder* is either **ASC** or **DESC**. Note that the *sortingOrder* does not have to be identical for each field. For example, the query:

```
SELECT AuthorID, FirstName, LastName
FROM Authors
ORDER BY LastName, FirstName
```

sorts all authors in ascending order by last name, then by first name. Thus, any authors have the same last name, their records are returned sorted by first name (Fig. 17.19).

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
6	Ted	Lin
12	Jonathan	Liperi
13	Jeffrey	Listfield
8	David	McPhie
3	Tem	Nieto
7	Praveen	Sadhu
5	Sean	Santry
4	Kate	Steinbuhler

Fig. 17.19 Authors from table **Authors** in ascending order by **LastName** and by **FirstName**. (Part 1 of 2.)

AuthorID	FirstName	LastName
11	Ben	Wiedermann
9	Cheryl	Yaeger
10	Marina	Zlatkina

Fig. 17.19 Authors from table **Authors** in ascending order by **LastName** and by **FirstName**. (Part 2 of 2.)

The **WHERE** and **ORDER BY** clauses can be combined in one query. For example, the query:

```
SELECT ISBN, Title, EditionNumber, Copyright, Price
FROM Titles
WHERE Title
LIKE '*How to Program' ORDER BY Title ASC
```

returns the ISBN, title, edition number, copyright and price of each book in the **Titles** table that has a **Title** ending with “**How to Program**;” it lists these records in ascending order by **Title**. The results of the query are depicted in Fig. 17.20.

ISBN	Title	Edition-Number	Copy-right	Price
0130895601	Advanced Java 2 Platform How to Program	1	2002	$69.95
0131180436	C How to Program	1	1992	$69.95
0130895725	C How to Program	3	2001	$69.95
0132261197	C How to Program	2	1994	$49.95
0130622214	C# How To Program	1	2002	$69.95
0135289106	C++ How to Program	2	1998	$49.95
0131173340	C++ How to Program	1	1994	$69.95
0130895717	C++ How to Program	3	2001	$69.95
013028419X	e-Business and e-Commerce How to Program	1	2001	$69.95
0130308978	Internet and World Wide Web How to Program	2	2002	$69.95
0130161438	Internet and World Wide Web How to Program	1	2000	$69.95
0130341517	Java How to Program	4	2002	$69.95
0136325890	Java How to Program	1	1998	$49.95

Fig. 17.20 Books from table **Titles** whose titles end with **How to Program** in ascending order by **Title**. (Part 1 of 2.)

ISBN	Title	Edition-Number	Copy-right	Price
0130284181	Perl How to Program	1	2001	$69.95
0130923613	Python How to Program	1	2002	$69.95
0130293636	Visual Basic .NET How to Program	2	2002	$69.95
0134569555	Visual Basic 6 How to Program	1	1999	$69.95
0130622265	Wireless Internet and Mobile Business How to Program	1	2001	$69.95
0130284173	XML How to Program	1	2001	$69.95

Fig. 17.20 Books from table **Titles** whose titles end with **How to Program** in ascending order by **Title**. (Part 2 of 2.)

17.4.4 Merging Data from Multiple Tables: INNER JOIN

Database designers often split related data into separate tables to ensure that a database does not store data redundantly. For example, the **Books** database has tables **Authors** and **Titles**. We use an **AuthorISBN** table to provide “links” between authors and their corresponding titles. If we did not separate this information into individual tables, we would need to include author information with each entry in the **Titles** table. This would result in the database storing duplicate author information for authors who wrote multiple books.

Often, it is necessary for analysis purposes to merge data from multiple tables into a single set of data—referred to as *joining* the tables. Joining is accomplished via an ***INNER JOIN*** operation in the **SELECT** query. An **INNER JOIN** merges records from two or more tables by testing for matching values in a field that is common to the tables. The simplest format for an **INNER JOIN** clause is:

```
SELECT fieldName1, fieldName2, …
FROM table1
INNER JOIN table2
   ON table1.fieldName = table2.fieldName
```

The **ON** part of the **INNER JOIN** clause specifies the fields from each table that are compared to determine which records are joined. For example, the following query produces a list of authors accompanied by the ISBN numbers for books written by each author:

```
SELECT FirstName, LastName, ISBN
FROM Authors
INNER JOIN AuthorISBN
   ON Authors.AuthorID = AuthorISBN.AuthorID
ORDER BY LastName, FirstName
```

The query merges the **FirstName** and **LastName** fields from table **Authors** with the **ISBN** field from table **AuthorISBN**, sorting the results in ascending order by **LastName** and **FirstName**. Notice the use of the syntax *tableName*.*fieldName* in the **ON** part of the **INNER JOIN**. This syntax (called a *fully qualified name*) specifies the fields from each ta-

ble that should be compared to join the tables. The "*tableName*`.`" syntax is required if the fields have the same name in both tables. The same syntax can be used in any query to distinguish among fields in different tables that have the same name. Fully qualified names that start with the database name can be used to perform cross-database queries.



Software Engineering Observation 17.1

If an SQL statement includes fields from multiple tables that have the same name, the statement must precede those field names with their table names and the dot operator (e.g., **`Authors.AuthorID`***).*



Common Programming Error 17.5

In a query, failure to provide fully qualified names for fields that have the same name in two or more tables is an error.

As always, the query can contain an **`ORDER BY`** clause. Figure 17.21 depicts the results of the preceding query, ordered by **`LastName`** and **`FirstName`**. [*Note*: To save space, we split the results of the query into two columns, each containing the **`FirstName`**, **`LastName`** and **`ISBN`** fields.]

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Harvey	Deitel	0130895601	Harvey	Deitel	0130829293
Harvey	Deitel	0130284181	Harvey	Deitel	0134569555
Harvey	Deitel	0130284173	Harvey	Deitel	0130829277
Harvey	Deitel	0130852473	Paul	Deitel	0130125075
Harvey	Deitel	0138993947	Paul	Deitel	0130856118
Harvey	Deitel	0130856118	Paul	Deitel	0130161438
Harvey	Deitel	0130161438	Paul	Deitel	013028419x
Harvey	Deitel	013028419x	Paul	Deitel	0139163050
Harvey	Deitel	0139163050	Paul	Deitel	0130895601
Harvey	Deitel	0135289106	Paul	Deitel	0135289106
Harvey	Deitel	0130895717	Paul	Deitel	0130895717
Harvey	Deitel	0132261197	Paul	Deitel	0132261197
Harvey	Deitel	0130895725	Paul	Deitel	0130895725
Harvey	Deitel	0130125075	Tem	Nieto	0130284181
Paul	Deitel	0130284181	Tem	Nieto	0130284173
Paul	Deitel	0130284173	Tem	Nieto	0130829293
Paul	Deitel	0130829293	Tem	Nieto	0134569555
Paul	Deitel	0134569555	Tem	Nieto	0130856118
Paul	Deitel	0130829277	Tem	Nieto	0130161438
Paul	Deitel	0130852473	Tem	Nieto	013028419x
Paul	Deitel	0138993947			

Fig. 17.21 Authors from table **`Authors`** and ISBN numbers of the authors' books, sorted in ascending order by **`LastName`** and **`FirstName`**.

17.4.5 Joining Data from Tables `Authors`, `AuthorISBN`, `Titles` and `Publishers`

The **`Books`** database contains one predefined query (**`TitleAuthor`**), which selects as its results the title, ISBN number, author's first name, author's last name, copyright year and publisher's name for each book in the database. For books that have multiple authors, the query produces a separate composite record for each author. The **`TitleAuthor`** query is depicted in Fig. 17.22. Figure 17.23 contains a portion of the query results.

```
SELECT Titles.Title, Titles.ISBN, Authors.FirstName,
       Authors.LastName, Titles.Copyright,
       Publishers.PublisherName
FROM
   ( Publishers INNER JOIN Titles
      ON Publishers.PublisherID = Titles.PublisherID )
   INNER JOIN
   ( Authors INNER JOIN AuthorISBN
      ON Authors.AuthorID = AuthorISBN.AuthorID )
   ON Titles.ISBN = AuthorISBN.ISBN
ORDER BY Titles.Title
```

Fig. 17.22 **`TitleAuthor`** query of **`Books`** database.

Title	ISBN	FirstName	LastName	Copyright	PublisherName
Advanced Java 2 Platform How to Program	0130895601	Paul	Deitel	2002	Prentice Hall
Advanced Java 2 Platform How to Program	0130895601	Harvey	Deitel	2002	Prentice Hall
Advanced Java 2 Platform How to Program	0130895601	Sean	Santry	2002	Prentice Hall
C How to Program	0131180436	Harvey	Deitel	1992	Prentice Hall
C How to Program	0131180436	Paul	Deitel	1992	Prentice Hall
C How to Program	0132261197	Harvey	Deitel	1994	Prentice Hall
C How to Program	0132261197	Paul	Deitel	1994	Prentice Hall
C How to Program	0130895725	Harvey	Deitel	2001	Prentice Hall
C How to Program	0130895725	Paul	Deitel	2001	Prentice Hall
C# How To Program	0130622214	Tem	Nieto	2002	Prentice Hall
C# How To Program	0130622214	Paul	Deitel	2002	Prentice Hall
C# How To Program	0130622214	Jeffrey	Listfield	2002	Prentice Hall
C# How To Program	0130622214	Cheryl	Yaeger	2002	Prentice Hall
C# How To Program	0130622214	Marina	Zlatkina	2002	Prentice Hall

Fig. 17.23 Portion of the result set produced by the query in Fig. 17.22. (Part 1 of 2.)

Title	ISBN	First-Name	Last-Name	Copy-right	Publisher-Name
C# How To Program	0130622214	Harvey	Deitel	2002	Prentice Hall
C++ How to Program	0130895717	Paul	Deitel	2001	Prentice Hall
C++ How to Program	0130895717	Harvey	Deitel	2001	Prentice Hall
C++ How to Program	0131173340	Paul	Deitel	1994	Prentice Hall
C++ How to Program	0131173340	Harvey	Deitel	1994	Prentice Hall
C++ How to Program	0135289106	Harvey	Deitel	1998	Prentice Hall
C++ How to Program	0135289106	Paul	Deitel	1998	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Harvey	Deitel	2000	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Kate	Stein-buhler	2000	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Paul	Deitel	2000	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Harvey	Deitel	2001	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Paul	Deitel	2001	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Tem	Nieto	2001	Prentice Hall

Fig. 17.23 Portion of the result set produced by the query in Fig. 17.22. (Part 2 of 2.)

We added indentation to the query in Fig. 17.22 to make the query more readable. Let us now break down the query into its various parts. Lines 1–3 contain a comma-separated list of the fields that the query returns; the order of the fields from left to right specifies the fields' order in the returned table. This query selects fields **`Title`** and **`ISBN`** from table **`Titles`**, fields **`FirstName`** and **`LastName`** from table **`Authors`**, field **`Copyright`** from table **`Titles`** and field **`PublisherName`** from table **`Publishers`**. For purposes of clarity, we fully qualified each field name with its table name (e.g., **`Titles.ISBN`**).

Lines 5–10 specify the **`INNER JOIN`** operations used to combine information from the various tables. There are three **`INNER JOIN`** operations. It is important to note that, although an **`INNER JOIN`** is performed on two tables, either of those two tables can be the result of another query or another **`INNER JOIN`**. We use parentheses to nest the **`INNER JOIN`** operations; SQL evaluates the innermost set of parentheses first and then moves outward. We begin with the **`INNER JOIN`**:

```
( Publishers INNER JOIN Titles
   ON Publishers.PublisherID = Titles.PublisherID )
```

which joins the **`Publishers`** table and the **`Titles`** table **`ON`** the condition that the **`PublisherID`** numbers in each table match. The resulting temporary table contains information about each book and its publisher.

The other nested set of parentheses contains the **INNER JOIN**:

```
( Authors INNER JOIN AuthorISBN ON
   Authors.AuthorID = AuthorISBN.AuthorID )
```

which joins the **Authors** table and the **AuthorISBN** table **ON** the condition that the **AuthorID** fields in each table match. Remember that the **AuthorISBN** table has multiple entries for **ISBN** numbers of books that have more than one author. The third **INNER JOIN**:

```
( Publishers INNER JOIN Titles
   ON Publishers.PublisherID = Titles.PublisherID )
INNER JOIN
( Authors INNER JOIN AuthorISBN
   ON Authors.AuthorID = AuthorISBN.AuthorID )
ON Titles.ISBN = AuthorISBN.ISBN
```

joins the two temporary tables produced by the two prior inner joins **ON** the condition that the **Titles.ISBN** field for each record in the first temporary table matches the corresponding **AuthorISBN.ISBN** field for each record in the second temporary table. The result of all these **INNER JOIN** operations is a temporary table from which the appropriate fields are selected to produce the results of the query.

Finally, line 11 of the query:

```
ORDER BY Titles.Title
```

indicates that all the records should be sorted in ascending order (the default) by title.

17.4.6 INSERT Statement

The ***INSERT*** statement inserts a new record in a table. The simplest form for this statement is:

```
INSERT INTO tableName ( fieldName1, fieldName2, …, fieldNameN )
   VALUES ( value1, value2, …, valueN )
```

where *tableName* is the table in which to insert the record. The *tableName* is followed by a comma-separated list of field names in parentheses. The list of field names is followed by the SQL keyword **VALUES** and a comma-separated list of values in parentheses. The specified values in this list must match the field names listed after the table name in both order and type (for example, if *fieldName1* is specified as the **FirstName** field, then *value1* should be a string in single quotes representing the first name). The **INSERT** statement:

```
INSERT INTO Authors ( FirstName, LastName )
   VALUES ( 'Sue', 'Smith' )
```

inserts a record into the **Authors** table. The statement indicates that values will be inserted for the **FirstName** and **LastName** fields. The corresponding values to insert are **'Sue'** and **'Smith'**. [*Note*: The SQL statement does not specify an **AuthorID** in this example, because **AuthorID** is an *autoincrement field* in table **Authors**. For every new record added to this table, MySQL assigns a unique **AuthorID** value that is the next value in the autoincrement sequence (i.e., 1, 2, 3, etc.). In this case, MySQL assigns **AuthorID** number 8 to Sue Smith.] Figure 17.24 shows the **Authors** table after the **INSERT INTO** operation.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield
14	Sue	Smith

Fig. 17.24 **Authors** after an **INSERT** operation to add a record.

Common Programming Error 17.6

*SQL statements use the single-quote (**'**) character as a delimiter for strings. To specify a string containing a single quote (such as O'Malley) in an SQL statement, the string must include two single quotes in the position where the single-quote character should appear in the string (e.g., **'O''Malley'**). The first of the two single-quote characters acts as an escape character for the second. Failure to escape single-quote characters in a string that is part of an SQL statement is an SQL syntax error.*

17.4.7 UPDATE Statement

An ***UPDATE*** statement modifies data in a table. The simplest form for an **UPDATE** statement is:

```
UPDATE tableName
   SET fieldName1 = value1, fieldName2 = value2, …, fieldNameN = valueN
   WHERE criteria
```

where *tableName* is the table in which to update a record (or records). The *tableName* is followed by keyword ***SET*** and a comma-separated list of field name/value pairs written in the format, *fieldName* **=** *value*. The **WHERE** clause specifies the criteria used to determine which record(s) to update. For example, the **UPDATE** statement:

```
UPDATE Authors
   SET LastName = 'Jones'
   WHERE LastName = 'Smith' AND FirstName = 'Sue'
```

updates a record in the **Authors** table. The statement indicates that **LastName** will be assigned the new value **Jones** for the record in which **LastName** currently is equal to

Smith and **FirstName** is equal to **Sue**. If we know the **AuthorID** in advance of the **UPDATE** operation (possibly because we searched for the record previously), the **WHERE** clause could be simplified as follows:

```
WHERE AuthorID = 14
```

Figure 17.25 depicts the **Authors** table after we perform the **UPDATE** operation.

Common Programming Error 17.7

Failure to use a ***WHERE*** *clause with an* ***UPDATE*** *statement could lead to logic errors.*

17.4.8 DELETE Statement

An SQL ***DELETE*** statement removes data from a table. The simplest form for a **DELETE** statement is:

```
DELETE FROM tableName WHERE criteria
```

where *tableName* is the table from which to delete a record (or records). The **WHERE** clause specifies the criteria used to determine which record(s) to delete. For example, the **DELETE** statement:

```
DELETE FROM Authors
   WHERE LastName = 'Jones' AND FirstName = 'Sue'
```

deletes the record for **Sue Jones** from the **Authors** table. Figure 17.26 depicts the **Authors** table after we perform the **DELETE** operation.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield
14	Sue	Jones

Fig. 17.25 Table **Authors** after an **UPDATE** operation to change a record.

`AuthorID`	`FirstName`	`LastName`
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

Fig. 17.26 Table **`Authors`** after a **`DELETE`** operation to remove a record.

Common Programming Error 17.8

`WHERE` *clauses can match multiple records. When deleting records from a database, be sure to define a* ***`WHERE`*** *clause that matches only the records to be deleted.*

17.5 Python DB-API Specification

The code examples in this chapter use the MySQL database system; however, Python supports many databases in addition to MySQL. Modules have been written that can interface with most popular databases, thus hiding database details from the programmer. These modules follow the Python *Database Application Programming Interface (DB-API)*, a document that specifies common object and method names for manipulating any database.

Specifically, the DB-API describes a ***`Connection`*** object that accesses the database (connects to the database). A program then uses the **`Connection`** object to create the ***`Cursor`*** object, which manipulates and retrieves data. We discuss the methods and attributes of these objects in the context of live-code examples throughout the remainder of the chapter.

A **`Cursor`** enables a program to perform operations on a database (e.g., executing queries, inserting rows into a table, deleting rows from a table, etc.), as well as manipulate data returned from query execution. Three methods are available to fetch row(s) of a query result set—**`fetchone`**, **`fetchmany`** and **`fetchall`**. Method **`fetchone`** returns a tuple containing the next row in a result set stored in **`Cursor`**. Method **`fetchmany`** takes one argument—the number of rows to be fetched and returns the next set of rows of a result set as a tuple of tuples. Method **`fetchall`** returns all rows of a result set as a tuple of tuples. On a large database, a **`fetchall`** would be impractical.

A benefit of the DB-API is that a program does not need to know much about the database to which the program connects. Therefore, a program can use different databases with few modifications in the Python source code. For example, to switch from the MySQL

database to another database, a programmer needs to change three or four lines of code. However, the switch between databases may require modifications to the SQL code (to compensate for case sensitivity, etc.).

17.6 Database Query Example

Figure 17.27 presents a CGI program that performs a simple query on the **Books** database. The query retrieves all information about the authors in the **Authors** table and displays the data in an XHTML table. The program demonstrates connecting to the database, querying the database and displaying the results. The discussion that follows presents the key DB-API aspects of the program. [Note: The CGI script in this example is defined for use with the Apache Web server running on Microsoft Windows. On the CD that accompanies this book, we provide a version of this example for use with Apache running on Linux.]

```
#!c:\python\python.exe
# Fig. 17.27: fig17_27.py
# Displays contents of the Authors table,
# ordered by a specified field.

import MySQLdb
import cgi
import sys

def printHeader( title ):
   print """Content-type: text/html

<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml"
   xml:lang = "en" lang = "en">
<head><title>%s</title></head>

<body>""" % title

# obtain user query specifications
form = cgi.FieldStorage()

# get "sortBy" value
if form.has_key( "sortBy" ):
   sortBy = form[ "sortBy" ].value
else:
   sortBy = "firstName"

# get "sortOrder" value
if form.has_key( "sortOrder" ):
   sortOrder = form[ "sortOrder" ].value
else:
   sortOrder = "ASC"

```

Fig. 17.27 Connecting to and querying a database and displaying the results.

```
printHeader( "Authors table from Books" )

# connect to database and retrieve a cursor
try:
   connection = MySQLdb.connect( db = "Books" )

# error connecting to database
except MySQLdb.OperationalError, error:
   print "Error:", error
   sys.exit( 1 )

# retrieve cursor
else:
   cursor = connection.cursor()

# query all records from Authors table
cursor.execute( "SELECT * FROM Authors ORDER BY %s %s" %
   ( sortBy, sortOrder ) )

allFields = cursor.description  # get field names
allRecords = cursor.fetchall()  # get records

# close cursor and connection
cursor.close()
connection.close()

# output results in a table
print """\n<table border = "1" cellpadding = "3" >
      <tr bgcolor = "silver" >"""

# create table header
for field in allFields:
   print "<td>%s</td>" % field[ 0 ]

print "</tr>"

# display each record as a row
for author in allRecords:
   print "<tr>"

   for item in author:
      print "<td>%s</td>" % item

   print "</tr>"

print "</table>"

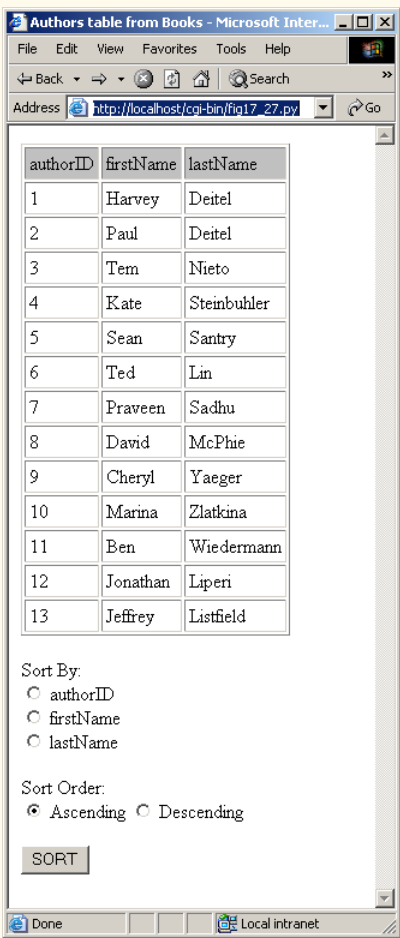
# obtain sorting method from user
print """
      \n<form method = "post" action = "/cgi-bin/fig17_27.py">
      Sort By:<br />"""

```

Fig. 17.27 Connecting to and querying a database and displaying the results.

```
# display sorting options
for field in allFields:
   print """<input type = "radio" name = "sortBy"
      value = "%s" />""" % field[ 0 ]
   print field[ 0 ]
   print "<br />"

print """<br />\nSort Order:<br />
      <input type = "radio" name = "sortOrder"
      value = "ASC" checked = "checked" />
      Ascending
      <input type = "radio" name = "sortOrder"
      value = "DESC" />
      Descending
      <br /><br />\n<input type = "submit" value = "SORT" />
      </form>\n\n</body>\n</html>"""
```



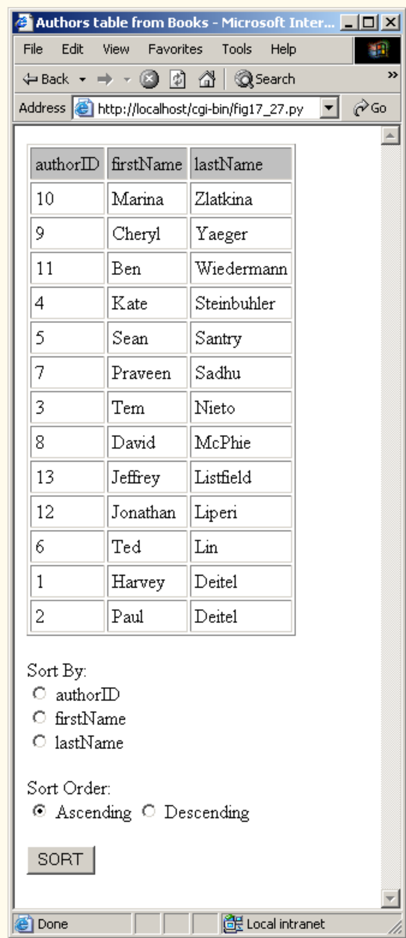


Fig. 17.27 Connecting to and querying a database and displaying the results.

Line 6 imports module ***MySQLdb***, which contains classes and functions for manipulating MySQL databases in Python (available from **sourceforge.net/projects/mysql-python**). For installation instructions, please visit **www.deitel.com**.

Lines 86–105 create an XHTML form that enables the user to specify how to sort the records of the **Authors** table. Lines 24–36 retrieve and process this form. The records are sorted by the field assigned to variable **sortBy**. By default, the records are sorted by **AuthorID**. The user can select a radio button to sort the records by another field. Similarly, variable **sortOrder** has either the user-specified value or **"ASC"**.

Line 42 creates a **Connection** object called **connection** to manage the connection between the program and the database. Function ***MySQLdb.connect*** receives the name of the database as the value of keyword argument **db** and creates the connection. [*Note*: For operating systems other than Windows, MySQL may require a username and password to connect to the database. If so, pass the appropriate values as strings to keyword arguments **user** and **passwd** for function **MySQLdb.connect**.] If **MySQLdb.connect** fails, the function raises a **MySQLdb** ***OperationalError*** exception.

Line 51 calls **Connection** method ***cursor*** to create a **Cursor** object for the database. The **Cursor** method ***execute*** takes as an argument an SQL command to execute against the database. Lines 54–55 query and retrieve all records from the **Authors** table sorted by the field specified in **sortBy** and ordered by the value of **sortOrder**.

A **Cursor** object internally stores the results of a database query. The **Cursor** attribute ***description*** contains a tuple of tuples in which each tuple provides information about a field in the result set obtained by method **execute**. The first value of each field's tuple is the field name. Line 57 assigns the tuple of field name records to variable **allFields**. Cursor method ***fetchall*** returns a tuple of tuples that contains all the internally stored results obtained by invoking method execute. Each subtuple in the returned tuple represents one record from the database, and each element in the record represents a field's value for that record. Line 58 assigns the tuple of matching records to variable **allRecords**.

Cursor method ***close*** (line 61) closes the **Cursor** object; line 62 closes the **Connection** object with **Connection** method ***close***. These methods explicitly close the **Cursor** and the **Connection** objects. Although the objects' **close** methods execute when the objects are destroyed at program termination, programmers should explicitly close the objects once they are no longer needed.

Good Programming Practice 17.2

*Explicitly close **Cursor** and **Connection** objects with their respective **close** methods as soon as the program no longer needs those objects.*

The remainder of the program displays the results of the database query in an XHTML table. Lines 65–83 display the **Authors** table's fields using a **for** loop. For each field, the program displays the first entry in that field's tuple (lines 69-70). Lines 75–83 display a table row for each record in the **Authors** table using nested **for** loops. The outer **for** loop (line 75) iterates through each record in the table to create a new row. The inner **for** loop (line 78) iterates over each field in the current record and displays each field in a new cell.

17.7 Querying the Books Database

Figure 17.28 enhances the example of Fig. 17.27 by allowing the user to enter any query into a GUI program. This example introduces database error handling and the **Pmw** compo-

nents **ScrolledFrame** and **PanedWidget**. Module **Pmw** is introduced in Chapter 11, Graphical User Interface Components: Part 2. The GUI constructor (lines 13–39) creates four GUI elements.

The program display contains two sections. The top section provides a **ScrolledText** component (lines 24–25) for entering a query string. The attribute **text_height** sets the scrolled text area as eight lines high. A **Button** component (lines 28–30) calls the method that executes the query string on the database.

The bottom section contains a ***ScrolledFrame*** component (lines 33–35) for displaying the results of the query. A **ScrolledFrame** component is a scrollable area. The horizontal and vertical scroll bars are displayed because attributes **hscrollmode** and **vscrollmode** are assigned the value **"static"**. The **ScrolledFrame** contains a ***PanedWidget*** component (lines 37–39) for dividing the result records into fields. **Frame** method **interior** specifies that the **PanedWidget** is created within the **ScrolledFrame**. A **PanedWidget** is a subdivided frame that allows the user to change the size of the subdivisions. The **PanedWidget** constructor's ***orient*** argument takes the value ***"horizontal"*** or ***"vertical"***. If the value is **"horizontal"**, the panes are placed left to right in the frame; if the value is **"vertical"**, the panes are placed top to bottom in the frame.

```
1  # Fig. 17.28: fig17_28.py
2  # Displays results returned by a
3  # query on Books database.
4
5  import MySQLdb
6  from Tkinter import *
7  from tkMessageBox import *
8  import Pmw
9
10 class QueryWindow( Frame ):
11    """GUI Database Query Frame"""
12
13    def __init__( self ):
14       """QueryWindow Constructor"""
15
16       Frame.__init__( self )
17       Pmw.initialise()
18       self.pack( expand = YES, fill = BOTH )
19       self.master.title( \
20          "Enter Query, Click Submit to See Results." )
21       self.master.geometry( "525x525" )
22
23       # scrolled text pane for query string
24       self.query = Pmw.ScrolledText( self, text_height = 8 )
25       self.query.pack( fill = X )
26
27       # button to submit query
28       self.submit = Button( self, text = "Submit query",
29          command = self.submitQuery )
30       self.submit.pack( fill = X )
```

Fig. 17.28 GUI application for submitting queries to a database. (Part 1 of 3.)

```

      # frame to display query results
      self.frame = Pmw.ScrolledFrame( self,
         hscrollmode = "static", vscrollmode = "static" )
      self.frame.pack( expand = YES, fill = BOTH )

      self.panes = Pmw.PanedWidget( self.frame.interior(),
         orient = "horizontal" )
      self.panes.pack( expand = YES, fill = BOTH )

   def submitQuery( self ):
      """Execute user-entered query agains database"""

      # open connection, retrieve cursor and execute query
      try:
         connection = MySQLdb.connect( db = "Books" )
         cursor = connection.cursor()
         cursor.execute( self.query.get() )
      except MySQLdb.OperationalError, message:
         errorMessage = "Error %d:\n%s" % \
            ( message[ 0 ], message[ 1 ] )
         showerror( "Error", errorMessage )
         return
      else:   # obtain user-requested information
         data = cursor.fetchall()
         fields = cursor.description   # metadata from query
         cursor.close()
         connection.close()

      # clear results of last query
      self.panes.destroy()
      self.panes = Pmw.PanedWidget( self.frame.interior(),
         orient = "horizontal" )
      self.panes.pack( expand = YES, fill = BOTH )

      # create pane and label for each field
      for item in fields:
         self.panes.add( item[ 0 ] )
         label = Label( self.panes.pane( item[ 0 ] ),
            text = item[ 0 ], relief = RAISED )
         label.pack( fill = X )

      # enter results into panes, using labels
      for entry in data:

         for i in range( len( entry ) ):
            label = Label( self.panes.pane( fields[ i ][ 0 ] ),
               text = str( entry[ i ] ), anchor = W,
               relief = GROOVE, bg = "white" )
            label.pack( fill = X )

      self.panes.setnaturalsize()

```

Fig. 17.28 GUI application for submitting queries to a database. (Part 2 of 3.)

```
84 def main():
85     QueryWindow().mainloop()
86
87 if __name__ == "__main__":
88     main()
```

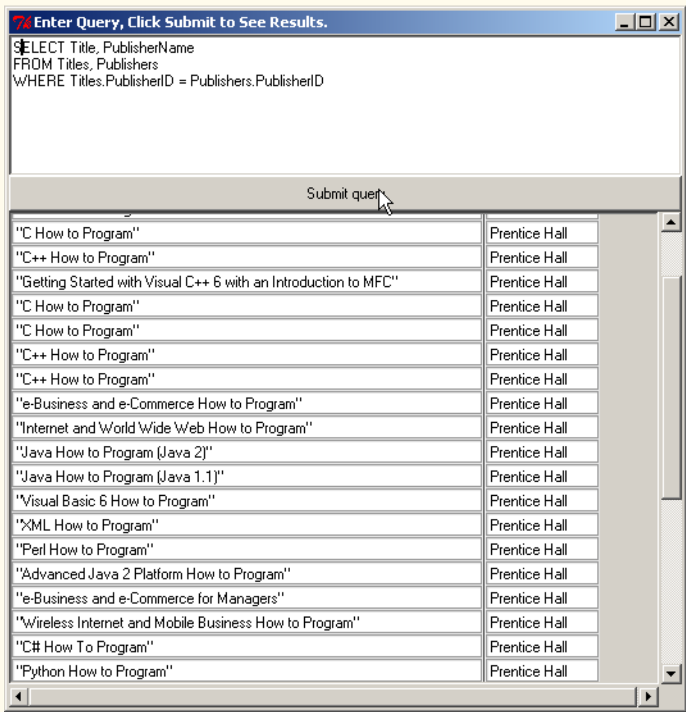


Fig. 17.28 GUI application for submitting queries to a database. (Part 3 of 3.)

When the user presses the **Submit query** button, method **submitQuery** (lines 41–82) performs the query and displays the results. Lines 45–58 contain a **try**/**except**/**else** statement that connects to and queries the database. The **try** statement creates a **Connection** and a **Cursor** object and uses **Cursor** method **execute** to perform the user-entered query. Function **MySQLdb.connect** (line 46) fails if the specified database does not exist. **Cursor** method **execute** (line 48) fails if the query string contains an SQL syntax error. Each method raises an **OperationalError** exception. Lines 49–53 handle this exception and call **tkMessageBox** function **showerror** with an appropriate error message.

If the user-entered query string successfully executes, the program retrieves the result of the query. The **else** clause (lines 54–58) assigns the queried records to variable **data** and assigns metadata to variable **fields**. *Metadata* is data that describes data. For example, the metadata for a result set may include the field names and field types. The metadata

```
fields = cursor.description
```

contains descriptive information about the result set of the user-entered query (line 56). **Cursor** attribute ***description*** contains a tuple of tuples that provides information about the fields obtained by method **execute**.

PanedWidget method ***destroy*** (line 61) removes the existing panes to display the query data in new panes (lines 62–64). Lines 67–71 iterate over the field information to display the names of the columns. For each field, method **add** adds a pane to the **PanedWidget**. This method takes a string that identifies the pane. The **Label** constructor adds a label to the pane that contains the name of the field with the **relief** attribute set to **RAISED**. **PanedWidget** method ***pane*** (line 69) identifies the parent of this new label. This method takes the name of a pane and returns a reference to that pane.

Lines 74–80 iterate over each record to create a label that contains the value of each field in the record. Method **pane** specifies the appropriate parent frame for each label. The expression

```
self.panes.pane( fields[ i ][ 0 ] )
```

evaluates to the pane whose name is the field name for the ith value in the record. Once the results have been added to the panes, the **PanedWidget** method ***setnaturalsize*** sets the size of each pane to be large enough to view the largest label in the pane.

17.8 Reading, Inserting and Updating a Database

The next example (Fig. 17.29) manipulates a simple MySQL **AddressBook** database that contains one table (**addresses**) with 11 columns—**ID** (a unique integer ID number for each person in the address book), **FirstName**, **LastName**, **Address**, **City**, **StateOrProvince**, **PostalCode**, **Country**, **EmailAddress**, **HomePhone** and **FaxNumber**. All fields, except **ID**, are strings. The program provides capabilities for inserting new records, updating existing records and searching for records in the database. [*Note*: The CD that accompanies this book contains a program called **DBSetup.py** that creates an empty **AddressBook** database.]

Class **AddressBook** uses **Button** and **Entry** components to retrieve and display address information. The constructor creates a list of fields for one address book entry (lines 32–34). Line 38 initializes dictionary data member **entries** to hold references to **Entry** components. A **for** loop (lines 44–60) then iterates over the length of this list to create an **Entry** component for each field (lines 47–48). The loop also adds a reference to the **Entry** component to data member **entries**. Lines 58–60 create a key name for each entry, based on that entry's field name.

```
1   # Fig. 17.29: fig17_29.py
2   # Inserts into, updates and searches a database
3
4   import MySQLdb
5   from Tkinter import *
6   from tkMessageBox import *
7   import Pmw
8
9   class AddressBook( Frame ):
10     """GUI Database Address Book Frame"""
```

Fig. 17.29 Inserting, finding and updating records. (Part 1 of 5.)

```

   def __init__( self ):
      """Address Book constructor"""

      Frame.__init__( self )
      Pmw.initialise()
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Address Book Database Application" )

      # buttons to execute commands
      self.buttons = Pmw.ButtonBox( self, padx = 0 )
      self.buttons.grid( columnspan = 2 )
      self.buttons.add( "Find", command = self.findAddress )
      self.buttons.add( "Add", command = self.addAddress )
      self.buttons.add( "Update", command = self.updateAddress )
      self.buttons.add( "Clear", command = self.clearContents )
      self.buttons.add( "Help", command = self.help, width = 14 )
      self.buttons.alignbuttons()


      # list of fields in an address record
      fields = [ "ID", "First name", "Last name",
        "Address", "City", "State Province", "Postal Code",
        "Country", "Email Address", "Home phone", "Fax Number" ]

      # dictionary with Entry components for values, keyed by
      # corresponding addresses table field names
      self.entries = {}

      self.IDEntry = StringVar()   # current address id text
      self.IDEntry.set( "" )

      # create entries for each field
      for i in range( len( fields ) ):
         label = Label( self, text = fields[ i ] + ":" )
         label.grid( row = i + 1, column = 0 )
         entry = Entry( self, name = fields[ i ].lower(),
            font = "Courier 12" )
         entry.grid( row = i + 1 , column = 1,
            sticky = W+E+N+S, padx = 5 )

         # user cannot type in ID field
         if fields[ i ] == "ID":
            entry.config( state = DISABLED,
               textvariable = self.IDEntry, bg = "gray" )

         # add entry field to dictionary
         key = fields[ i ].replace( " ", "_" )
         key = key.upper()
         self.entries[ key ] = entry

   def addAddress( self ):
      """Add address record to database"""
```

Fig. 17.29 Inserting, finding and updating records. (Part 2 of 5.)

```

      if self.entries[ "LAST_NAME" ].get() != "" and \
         self.entries[ "FIRST_NAME"].get() != "":

         # create INSERT query command
         query = """INSERT INTO addresses (
                 FIRST_NAME, LAST_NAME, ADDRESS, CITY,
                 STATE_PROVINCE, POSTAL_CODE, COUNTRY,
                 EMAIL_ADDRESS, HOME_PHONE, FAX_NUMBER
                 ) VALUES (""" + \
                 "'%s', " * 10 % \
                 ( self.entries[ "FIRST_NAME" ].get(),
                   self.entries[ "LAST_NAME" ].get(),
                   self.entries[ "ADDRESS" ].get(),
                   self.entries[ "CITY" ].get(),
                   self.entries[ "STATE_PROVINCE" ].get(),
                   self.entries[ "POSTAL_CODE" ].get(),
                   self.entries[ "COUNTRY" ].get(),
                   self.entries[ "EMAIL_ADDRESS" ].get(),
                   self.entries[ "HOME_PHONE" ].get(),
                   self.entries[ "FAX_NUMBER" ].get() )
         query = query[ :-2 ] + ")"

         # open connection, retrieve cursor and execute query
         try:
            connection = MySQLdb.connect( db = "AddressBook" )
            cursor = connection.cursor()
            cursor.execute( query )
         except MySQLdb.OperationalError, message:
            errorMessage = "Error %d:\n%s" % \
               ( message[ 0 ], message[ 1 ] )
            showerror( "Error", errorMessage )
         else:
            cursor.close()
            connection.close()
            self.clearContents()

      else:   # user has not filled out first/last name fields
         showwarning( "Missing fields", "Please enter name" )

   def findAddress( self ):
      """Query database for address record and display results"""

      if self.entries[ "LAST_NAME" ].get() != "":

         # create SELECT query
         query = "SELECT * FROM addresses " + \
                 "WHERE LAST_NAME = '" + \
                 self.entries[ "LAST_NAME" ].get() + "'"

         # open connection, retrieve cursor and execute query
         try:
            connection = MySQLdb.connect( db = "AddressBook" )
```

Fig. 17.29 Inserting, finding and updating records. (Part 3 of 5.)

```
             cursor = connection.cursor()
             cursor.execute( query )
          except MySQLdb.OperationalError, message:
             errorMessage = "Error %d:\n%s" % \
                ( message[ 0 ], message[ 1 ] )
             showerror( "Error", errorMessage )
             self.clearContents()
          else:   # process results
             results = cursor.fetchall()
             fields = cursor.description

             if not results:   # no results for this person
                showinfo( "Not found", "Nonexistent record" )
             else:             # display information in GUI
                self.clearContents()

                # display results
                for i in range( len( fields ) ):

                   if fields[ i ][ 0 ] == "ID":
                      self.IDEntry.set( str( results[ 0 ][ i ] ) )
                   else:
                      self.entries[ fields[ i ][ 0 ] ].insert(
                         INSERT, str( results[ 0 ][ i ] ) )

             cursor.close()
             connection.close()

       else:   # user did not enter last name
          showwarning( "Missing fields", "Please enter last name" )

    def updateAddress( self ):
       """Update address record in database"""

       if self.entries[ "ID" ].get():

          # create UPDATE query command
          entryItems= self.entries.items()
          query = "UPDATE addresses SET"

          for key, value in entryItems:

             if key != "ID":
                query += " %s='%s'," % ( key, value.get() )

          query = query[ :-1 ] + " WHERE ID=" + self.IDEntry.get()

          # open connection, retrieve cursor and execute query
          try:
             connection = MySQLdb.connect( db = "AddressBook" )
             cursor = connection.cursor()
             cursor.execute( query )
```

Fig. 17.29 Inserting, finding and updating records. (Part 4 of 5.)

```
         except MySQLdb.OperationalError, message:
            errorMessage = "Error %d:\n%s" % \
               ( message[ 0 ], message[ 1 ] )
            showerror( "Error", errorMessage )
            self.clearContents()
         else:
            showinfo( "database updated", "Database Updated." )
            cursor.close()
            connection.close()

      else:   # user has not specified ID
         showwarning( "No ID specified", """
            You may only update an existing record.
            Use Find to locate the record,
            then modify the information and press Update.""" )

   def clearContents( self ):
      """Clear GUI panel"""

      for entry in self.entries.values():
         entry.delete( 0, END )

      self.IDEntry.set( "" )

   def help( self ):
      "Display help message to user"

      showinfo( "Help", """Click Find to locate a record.
         Click Add to insert a new record.
         Click Update to update the information in a record.
         Click Clear to empty the Entry fields.\n""" )

def main():
   AddressBook().mainloop()

if __name__ == "__main__":
   main()
```

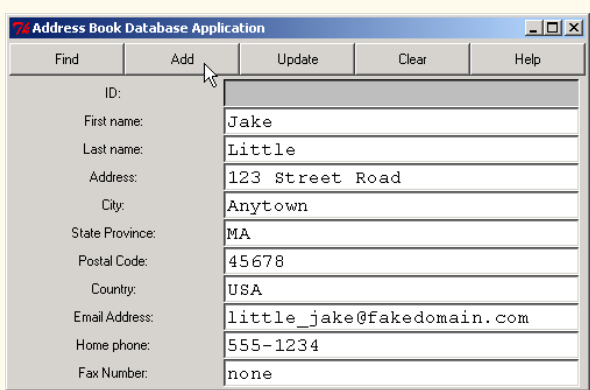


Fig. 17.29 Inserting, finding and updating records. (Part 5 of 5.)

Method **addRecord** (lines 62–102) adds a new record to the **AddressBook** database in response to the **Add** button in the GUI. The method first ensures that the user has entered values for the first and last name fields (lines 65–66). If the user enters values for these fields, the query string inserts a record into the database (lines 69–85). Otherwise, **tkMessageBox** function **showwarning** reminds the user to enter the information (lines 101–102). Line 74 includes ten string escape sequences whose values are replaced by the values contained in lines 75–84. Line 85 closes the values parentheses in the SQL statement.

Lines 88–99 contain a **try**/**except**/**else** statement that connects to and updates the database (i.e., inserts the new record in the database). Line 99 invokes method **clearContents** (lines 185–191) to clear the contents of the GUI. If an error occurs, **tkMessageBox** function **showerror** displays the error.

Method **findAddress** (lines 104–146) queries the **AddressBook** database for a specific record when the user clicks the **Find** button in the GUI. Line 107 tests whether the last name text field contains data. If the entry is empty, the program displays an error. If the user has entered data in the last name text field, a **SELECT** SQL statement searches the database for the user-specified last name. We used asterisk (*) in the **SELECT** statement because line 126 uses metadata to get field names. Lines 115–143 contain a **try**/**except**/**else** statement that connects to and queries the database. If these operations succeed, the program retrieves the results from the database (lines 125–126). A message informs the user if the query does not yield results (lines 128–129). If the query does yield results, lines 134–140 display the results in the GUI. Each field value is inserted in the appropriate **Entry** component. The record's ID must be converted to a string before it can be displayed.

Method **updateAddress** (lines 148–183) updates an existing database record. The program displays a message if the user attempts to perform an update operation on a nonexistent record. Line 151 tests whether the **id** for the current record is valid. Lines 155–162 create the SQL **UPDATE** statement. Lines 165–177 connect to and update the database.

Method **clearContents** (lines 185–191) clears the text fields when the user clicks the **Clear** button in the GUI. Method **help** (lines 193–199) calls a **tkMessageBox** function to display instructions about how to use the program.

17.9 Internet and World Wide Web Resources

This section presents several Internet and World Wide Web resources related to database programming.

www.mysql.com
This site offers the free MySQL database for download, the most current documentation and information about open-source licensing.

ww3.one.net/~jhoffman/sqltut.html
The *Introduction to SQL* has a tutorial, links to sites with more information about the language and examples.

www.python.org/topics/databases
This **python.org** page has links to modules like **MySQLdb**, documentation, a list of useful books about database programming and the DB-API specification.

www.chordate.com/gadfly.html
Gadfly is a free relational database written completely in Python. From this home page, visitors can download the database and view its documentation.

SUMMARY

- A database is an integrated collection of data.
- A database management system (DBMS) provides mechanisms for storing and organizing data in a manner consistent with the database's format. Database management systems allow for the access and storage of data without worrying about the internal representation of databases.
- Today's most popular database systems are relational databases.
- A language called Structured Query Language (SQL—pronounced as its individual letters or as "sequel") is used almost universally with relational database systems to perform queries (i.e., to request information that satisfies given criteria) and to manipulate data.
- Python programmers communicate with databases using modules that conform to the Python Database Application Programming Interface (DB-API).
- The relational database model is a logical representation of data that allows the relationships between the data to be considered independent of the actual physical structure of the data.
- A relational database is composed of tables. Any particular row of the table is called a record (or row).
- A primary key is a field (or fields) in a table that contain(s) unique data, which cannot be duplicated in other records. This guarantees each record can be identified by a unique value.
- A foreign key is a field in a table for which every entry has a unique value in another table and where the field in the other table is the primary key for that table. The foreign key helps maintain the Rule of Referential Integrity—every value in a foreign-key field must appear in another table's primary-key field. Foreign keys enable information from multiple tables to be joined together and presented to the user.
- Each column of the table represents a different field (or column or attribute). Records normally are unique (by primary key) within a table, but particular field values may be duplicated between records.
- SQL enables programmers to define complex queries that select data from a table by providing a complete set of commands.
- The results of a query commonly are called result sets (or record sets).
- A typical SQL query selects information from one or more tables in a database. Such selections are performed by **`SELECT`** queries. The simplest format of a **`SELECT`** query is

```
SELECT * FROM tableName
```

- An asterisk (**`*`**) indicates that all rows and columns from table *tableName* of the database should be selected.
- To select specific fields from a table, replace the asterisk (**`*`**) with a comma-separated list of field names.
- In most cases, it is necessary to locate records in a database that satisfy certain selection criteria. Only records that match the selection criteria are selected. SQL uses the optional **`WHERE`** clause in a **`SELECT`** query to specify the selection criteria for the query. The simplest format of a **`SELECT`** query with selection criteria is

```
SELECT fieldName1 FROM tableName WHERE criteria
```

- The **`WHERE`** clause condition can contain operators **`<`**, **`>`**, **`<=`**, **`>=`**, **`=`**, **`<>`** and **`LIKE`**.
- Operator **`LIKE`** is used for pattern matching with wildcard characters percent (**`%`**) and underscore (_). Pattern matching allows SQL to search for similar strings that "match a pattern."

- A pattern that contains a percent character (**%**) searches for strings that have zero or more characters at the percent character's position in the pattern.
- An underscore (_) in the pattern string indicates a single character at that position in the pattern.
- The results of a query can be arranged in ascending or descending order using the optional **ORDER BY** clause. The simplest form of an **ORDER BY** clause is

```
SELECT * FROM tableName ORDER BY field ASC
SELECT * FROM tableName ORDER BY field DESC
```

  where **ASC** specifies ascending order (lowest to highest), **DESC** specifies descending order (highest to lowest) and field specifies the field on which the sort is based.
- Multiple fields can be used for ordering purposes with an **ORDER BY** clause of the form

```
ORDER BY field1 sortingOrder, field2 sortingOrder, …
```

  where *sortingOrder* is either **ASC** or **DESC**. Note that the *sortingOrder* does not have to be identical for each field.
- The **WHERE** and **ORDER BY** clauses can be combined in one query.
- A join merges records from two or more tables by testing for matching values in a field that is common to both tables. The simplest format of a join is

```
SELECT fieldName1, fieldName2, …
FROM table1
INNER JOIN table2
   ON table1.fieldName = table2.fieldName
```

- A fully qualified name specifies the fields from each table that should be compared to join the tables. The "*tableName***.**" syntax is required if the fields have the same name in both tables. The same syntax can be used in a query to distinguish fields in different tables that happen to have the same name. Fully qualified names that start with the database name can be used to perform cross-database queries.
- The **INSERT** statement inserts a new record in a table. The simplest form of this statement is

```
INSERT INTO tableName ( fieldName1, …, fieldNameN )
VALUES ( value1,…, valueN )
```

  where *tableName* is the table in which to insert the record. The *tableName* is followed by a comma-separated list of field names in parentheses. (This list is not required if the **INSERT INTO** operation specifies a value for every column of the table in the correct order.) The list of field names is followed by the SQL keyword **VALUES** and a comma-separated list of values in parentheses. The values specified here should match the field names specified after the table name in order and type (i.e., if *fieldName1* is supposed to be the **FirstName** field, then *value1* should be a string in single quotes representing the first name).
- An **UPDATE** statement modifies data in a table. The simplest form for an **UPDATE** statement is

```
UPDATE tableName
SET fieldName1 = value1, …, fieldNameN = valueN
WHERE criteria
```

  where *tableName* is the table in which to update a record (or records). The *tableName* is followed by keyword **SET** and a comma-separated list of field name/value pairs in the format *fieldName* **=** *value*. The **WHERE** clause specifies the criteria used to determine which record(s) to update.

- An SQL **DELETE** statement removes data from a table. The simplest form for a **DELETE** statement is

  DELETE FROM *tableName* **WHERE** *criteria*

  where *tableName* is the table from which to delete a record (or records). The **WHERE** clause specifies the criteria used to determine which record(s) to delete.
- Modules have been written that can interface with most popular databases, hiding database details from the programmer. These modules follow the Python Database Application Programming Interface (DB-API), a document that specifies common object and method names for manipulating any database.
- The DB-API describes a **Connection** object that programs create to connect to a database.
- A program can use a **Connection** object to create a **Cursor** object, which the program uses to execute queries against the database.
- The major benefit of the DB-API is that a program does not need to know much about the database to which the program connects. Therefore, the programmer can change the database a program uses without changing vast amounts of Python code. However, changing the DB often requires changes in the SQL code.
- Module **MySQLdb** contains classes and functions for manipulating MySQL databases in Python.
- Function **MySQLdb.connect** creates the connection. The function receives the name of the database as the value of keyword argument **db**. If **MySQLdb.connect** fails, the function raises an **OperationalError** exception.
- The **Cursor** method **execute** takes as an argument a query string to execute against the database.
- A **Cursor** object internally stores the results of a database query.
- The **Cursor** method **fetchall** returns a tuple of records that matched the query. Each record is represented as a tuple that contains the values of that records field.
- The **Cursor** method **close** closes the **Cursor** object.
- The **Connection** method **close** closes the **Connection** object.
- A **PanedWidget** is a subdivided frame that allows the user to change the size of the subdivisions. The **PanedWidget** constructor's **orient** argument takes the value **"horizontal"** or **"vertical"**. If the value is **"horizontal"**, the panes are placed left to right in the frame; if the value is **"vertical"**, the panes are placed top to bottom in the frame.
- Metadata are data that describe other data. The **Cursor** attribute **description** contains a tuple of tuples that provides information about the fields of the data obtained by function **execute**. The cursor and connection are closed.
- The **PanedWidget** method **pane** takes the name of a pane and returns a reference to that pane.
- The **PanedWidget** method **setnaturalsize** sets the size of each pane to be large enough to view the largest label in the pane.

TERMINOLOGY

AND keyword
ASC keyword
asterisk (*****)
close method
column
Connection object
Cursor object
data attribute
database
database management system (DBMS)

database table
DELETE statement
DESC keyword
escape character
execute method
fetchall method
field
foreign key
FROM keyword
fully qualified name
INSERT statement
INTO keyword
interior method
joining tables
LIKE keyword
MySQL
MySQLdb module
open source
ORDER BY keyword
PanedWidget
pattern matching
percent (**%**) SQL wildcard character
primary key
Python Database Application Programming Interface (DB-API)
query
record
record set
relational database
result set
row
Rule of Referential Integrity
scalability
ScrolledFrame component
SELECT statement
selection criteria
SET keyword
shell
Structured Query Language (**SQL**)
table
underscore (_) wildcard character
UPDATE statement
VALUES keyword
WHERE clause
percent

SELF-REVIEW EXERCISES

17.1 Fill in the blanks in each of the following statements:

a) The most popular database query language is __________.
b) A relational database is composed of __________.
c) A table in a database consists of __________ and __________.
d) The __________ uniquely identifies each record in a table.
e) SQL provides a complete set of commands (including **SELECT**) that enable programmers to define complex __________.
f) SQL keyword __________ is followed by the selection criteria that specify the records to select in a query.
g) SQL keyword __________ specifies the order in which records are sorted in a query.
h) A __________ specifies the fields from multiple tables table that should be compared to join the tables.
i) A __________ is an integrated collection of data which is centrally controlled.
j) A __________ is a field in a table for which every entry has a unique value in another table and where the field in the other table is the primary key for that table.

17.2 State whether the following are *true* or *false*. If *false*, explain why.

a) **DELETE** is not a valid SQL keyword.
b) Tables in a database must have a primary key.
c) Python programmers communicate with databases using modules that conform to the DB-API.
d) **UPDATE** is a valid SQL keyword.
e) The **WHERE** clause condition can not contain operator **<>**.
f) Not all database systems support the **LIKE** operator.
g) The **INSERT INTO** statement inserts a new record in a table.
h) **MySQLdb.connect** is used to create a connection to database.

i) A **Cursor** object can execute queries in a database.
j) Once created, a connection with database can not be closed.

ANSWERS TO SELF-REVIEW EXERCISES

17.1 a) SQL. b) tables. c) rows, columns. d) primary key. e) queries. f) **WHERE**. g) **ORDER BY**. h) fully qualified name. i) database. j) foreign key.

17.2 a) False. **DELETE** is a valid SQL keyword—it is used to delete records. b) False. Tables in a database normally have primary keys. c) True. d) True. e) False. The **WHERE** clause can contain operator **<>** (not equals). f) True. g) True. h) True. i) True. j) False. **Connection.close** can close the connection.

EXERCISES

17.3 Write SQL queries for the **Books** database (discussed in Section 17.3) that perform each of the following tasks:

a) Select all authors from the **Authors** table.
b) Select all publishers from the **Publishers** table.
c) Select a specific author and list all books for that author. Include the title, copyright year and ISBN number. Order the information alphabetically by title.
d) Select a specific publisher and list all books published by that publisher. Include the title, copyright year and ISBN number. Order the information alphabetically by title.

17.4 Write SQL queries for the **Books** database (discussed in Section 17.3) that perform each of the following tasks:

a) Add a new author to the **Authors** table.
b) Add a new title for an author (remember that the book must have an entry in the **AuthorISBN** table). Be sure to specify the publisher of the title.
c) Add a new publisher.

17.5 Modify Fig. 17.27 so that the user can read different tables in the books database.

17.6 Create a MySQL database that contains information about students in a university. Possible fields might include date of birth, major, current grade point average, credits earned, etc. Write a Python program to manage the database. Include the following functionality: sort all students according to GPA (descending), create a display of all students in one particular major and remove all records from the database where the student has the required amount of credits to graduate.

17.7 Modify the **FIND** capability in Fig. 17.29 to allow the user to scroll through the results of the query in case there is more than one person with the specified last name in the Address Book. Provide an appropriate GUI.

17.8 Modify the solution from Exercise 17.7 so that the program checks whether a record already exists in the database before adding it.

19

Multithreading

Objectives

- To understand the notion of multithreading.
- To appreciate how multithreading can improve performance.
- To understand how to create, manage and destroy threads.
- To understand the life cycle of a thread.
- To study several examples of thread synchronization.
- To understand daemon threads.

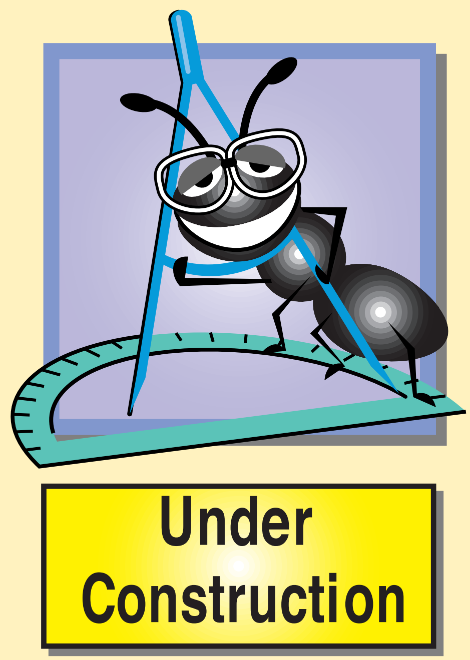
The spider's touch, how exquisitely fine!
Feels at each thread, and lives along the line.
Alexander Pope

A person with one watch knows what time it is; a person with two watches is never sure.
Proverb

Conversation is but carving!
Give no more to every guest,
Then he's able to digest.
Jonathan Swift

Learn to labor and to wait.
Henry Wadsworth Longfellow

The most general definition of beauty…Multeity in Unity.
Samuel Taylor Coleridge



Outline

19.1 Introduction

In Chapter 18, we discussed how to use processes to perform concurrent tasks in our programs. In this chapter, we discuss *multithreading* techniques for performing similar tasks. A thread is often called a “light-weight” process, because the operating system generally requires less resources to create and manage threads.

Python is different than many popular general-purpose programming languages in that it makes multithreading primitives available to the applications programmer. The programmer specifies that applications contain threads of execution, each thread designating a portion of a program that may execute concurrently with other threads. This capability gives the Python programmer powerful capabilities not available in C, C++ or other single-threaded languages.

Many tasks require a multithreaded programming approach. When a browser downloads large files such as audio clips or video clips from the World Wide Web, we do not want to wait until an entire clip is downloaded before starting the playback. So we can put multiple threads to work: one that downloads a clip, and another that plays the clip so that these activities, or tasks, may proceed concurrently. To avoid choppy playback, we coordinate the threads so that the player thread does not begin until there is a sufficient amount of the clip in memory to keep the player thread busy.

Performance Tip 19.1

A problem with single-threaded applications is that possibly lengthy activities must complete before other activities can begin. Users feel they already spend too much time waiting with Internet and World Wide Web applications, so multithreading is immediately appealing.

Another example of multithreading is Python’s automatic garbage collection. C and C++ place the responsibility for reclaiming dynamically allocated memory with the programmer. Python provides a garbage collector thread that automatically reclaims memory that is no longer needed.

Testing and Debugging Tip 19.1

In C and C++, programmers must explicitly provide statements for reclaiming dynamically allocated memory. When memory is not reclaimed (because a programmer forgets to do so, or because of a logic error or because an exception diverts program control), this results in an all-too-common error called a memory leak that can eventually exhaust the supply of free memory and may cause program termination. Python's automatic garbage collection eliminates the vast majority of memory leaks, i.e., those that are due to orphaned (unreferenced) objects.

Performance Tip 19.2

Python's garbage collection is not as efficient as the dynamic memory management code the best C and C++ programmers write, but it is relatively efficient and much safer for the programmer.

Performance Tip 19.3

Setting an object reference to ***`None`*** *marks that object for eventual garbage collection (if there are no other references to the object). This can help conserve memory in a system in which a local object is not going out of scope because the method it is in executes for a lengthy period.*

Writing multithreaded programs can be tricky. Although the human mind can perform many functions concurrently, humans find it difficult to jump between parallel "trains of thought." To see why multithreading can be difficult to program and understand, try the following experiment: Open three books to page 1. Now try reading the books concurrently. Read a few words from the first book, then read a few words from the second book, then read a few words from the third book, then loop back and read the next few words from the first book, and so on. After a brief time you rapidly appreciate the challenges of multithreading: switching between books, reading briefly, remembering your place in each book, moving the book you are reading closer so you can see it, pushing books you are not reading aside, and amidst all this chaos, trying to comprehend the content of the books!

19.2 `threading` Module

In this section we overview the various thread-related Python capabilities provided by *module* ***`threading`***. Although Python is perhaps one of the most portable programming languages, certain portions of the language are nevertheless platform dependent. The default installation of Python may not install the **`threading`** module on all systems. In this case, the **`threading`** module may need to be compiled by hand and Python re-installed.

Portability Tip 19.1

Python multithreading is platform dependent. Thus, the **`threading`** *module may have to be compiled by hand and reinstalled with Python.*

Programs create threads by instantiating objects of class ***`threading.Thread`***. Usually, we create a subclass of class **`Thread`** that extends the basic capabilities of the class to perform the tasks we want to perform. The code that "does the real work" of a thread is placed in its ***`run`*** method. The **`run`** method is overridden in a subclass of **`Thread`**.

A program launches a thread's execution by calling the thread's ***start*** method, which, in turn, calls the **run** method. After **start** launches the thread, **start** returns to its caller immediately. The caller then executes concurrently with the launched thread. If the thread has already been started, the **start** method raises an **AssertionError** exception.

Method ***isAlive*** returns 1 if **start** has been called for a given thread and the thread is not dead (i.e., its controlling **run** method has not completed execution). Method ***setName*** sets a **Thread**'s name. Method ***getName*** returns the name of the **Thread**. Using the **print** statement on a thread displays the thread's name and current state. Function ***threading.currentThread*** returns a reference to the currently executing **Thread**. Function ***threading.enumerate*** returns a list of all currently executing **Thread** objects, including the main thread. Function ***threading.activeCount*** returns the length of the list returned by **threading.enumerate**.

Thread method ***join*** waits for the thread whose **join** method is called to die before the caller can proceed. A thread may not call its own **join** method, only that of other threads. An optional argument accepted by **join** is a *timeout*, a floating-point number specifying the number of seconds that the caller waits. Passing no argument to method **join** indicates that the caller waits forever for the target thread to die before the caller proceeds. Such waiting can be dangerous; it can lead to two particularly serious problems called *deadlock*, in which one or more threads will wait forever for an event that cannot occur, and *indefinite postponement*, in which one or more threads will be delayed for some unpredictably long time. We will discuss deadlock in more detail in Section 19.5.

19.3 Thread Scheduling

The Python interpreter controls all threads in a program. When the interpreter starts, either in an interactive session or when invoked on a file, the "main" thread begins. This thread is the caller for all other threads. Only one thread is permitted to run by the interpreter at any one time. The interpreter keeps track of the *global interpreter lock (GIL)* that controls which thread the interpreter is running. When a program contains more than one running thread, these threads are *switched* in and out of the interpreter through the GIL, at specified intervals.

19.4 Thread States: Life Cycle of a Thread

At any time, a thread is said to be in one of several *thread states* (Fig. 19.1). Let us say that a thread that was just created is in the *born* state. The thread remains in this state until the thread's **start** method is called; this causes the thread to enter the *ready* state (also known as the *runnable* state). A *ready* thread enters the *running* state when the interpreter executes the thread (i.e. method **run** executes). A thread enters the *dead* state when its **run** method

completes or terminates for any reason—the interpreter eventually disposes of a *dead* thread.

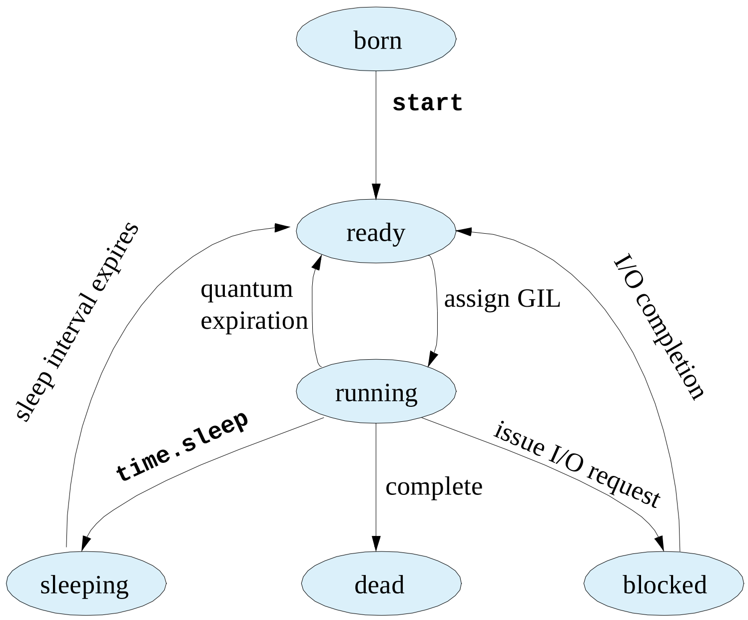


Fig. 19.1 Life cycle of a thread.

One common way for a *running* thread to enter the *blocked* state is when the thread issues an input/output request. In this case, a blocked thread becomes ready when the I/O it is waiting for completes. The interpreter does not execute a blocked thread even if the interpreter is free.

When a running thread calls function **`time.sleep`**, that thread enters the *sleeping* state. A sleeping thread becomes ready after the designated sleep time expires. A sleeping thread cannot use the interpreter. A thread enters the *dead* state when its **`run`** method either completes or raises an uncaught exception.

The program in Fig. 19.2 demonstrates basic threading techniques, including creation of a class derived from **`threading.Thread`**, construction of a thread and using function **`time.sleep`** in a thread. Each thread of execution created in the program displays its name after sleeping for a random amount of time between 1 and 5 seconds.

```
# Fig. 19.2: fig19_02.py
# Show multiple threads printing at different intervals.

import threading
import random
import time

```

Fig. 19.2 Multiple threads printing at random intervals (part 1 of 2).

```
class PrintThread( threading.Thread ):
   """Subclass of threading.Thread"""

   def __init__( self, threadName ):
      """Initialize thread, set sleep time, print data"""

      threading.Thread.__init__( self, name = threadName )
      self.sleepTime = random.randrange( 1, 6 )
      print "Name: %s; sleep: %d" % \
         ( self.getName(), self.sleepTime )

   # overridden Thread run method
   def run( self ):
      """Sleep for 1-5 seconds"""

      print self.getName(), "going to sleep"
      time.sleep( self.sleepTime )
      print self.getName(), "done sleeping"

thread1 = PrintThread( "thread1" )
thread2 = PrintThread( "thread2" )
thread3 = PrintThread( "thread3" )
thread4 = PrintThread( "thread4" )

print "\nStarting threads"

thread1.start()   # invokes run method of thread1
thread2.start()   # invokes run method of thread2
thread3.start()   # invokes run method of thread3
thread4.start()   # invokes run method of thread4

print "Threads started\n"
```

```
Name: thread1; sleep: 5
Name: thread2; sleep: 3
Name: thread3; sleep: 4
Name: thread4; sleep: 1

Starting threads
thread1 going to sleep
thread2 going to sleep
thread3 going to sleep
thread4 going to sleep
Threads started

thread4 done sleeping
thread2 done sleeping
thread3 done sleeping
thread1 done sleeping
```

Fig. 19.2 Multiple threads printing at random intervals (part 2 of 2).

Class **PrintThread**—which inherits from **threading.Thread** so each object of the class can execute in parallel—consists of attribute **sleepTime**, a constructor and a

run method. Attribute **sleepTime** stores a random integer value determined when a **PrintThread** object is constructed. When started, each **PrintThread** object sleeps for the amount of time specified by **sleepTime**, and then outputs its name.

The **PrintThread** constructor (lines 11–17) first calls the base class constructor. passing the class instance and the thread's name. A thread's name is specified with **Thread** keyword argument ***name***. If no name is specified, the thread will be assigned a unique name in the form **"Thread-*n*"** where *n* is an integer. The constructor then initializes **sleepTime** to a random integer between 1 and 5, inclusive. Then, the program outputs the name of the thread and the value of **sleepTime**, to show the values for the particular **PrintThread** being constructed.

When a **PrintThread**'s **start** method (inherited from **Thread**) is invoked, the **PrintThread** object enters the *ready* state. When the interpreter switches in the **PrintThread** object, it enters the *running* state and its **run** method begins execution. Method **run** (lines 20–25) prints a message indicating that the thread is going to sleep and then invokes function **time.sleep** (line 24) to immediately put the thread into a *sleeping* state. When the thread awakens after **sleepTime** seconds, it is placed into a *ready* state again until it is switched into the processor. When the **PrintThread** object enters the *running* state again, it outputs its name (indicating that the thread is done sleeping), its **run** method terminates and the thread object enters the *dead* state.

The main portion of the program instantiates four **PrintThread** objects and invokes the **Thread** class **start** method on each one to place all four **PrintThread** objects in a *ready* state. After this, the main program's thread terminates. However, the example continues running until the last **PrintThread** dies (i.e., has completed its **run** method).

19.5 Thread Synchronization

Multithreaded programs often contain code wherein two or more threads attempt to access and/or modify the value of a shared piece of data. For example, two threads may be reading and updating the value of a variable simultaneously. If a multithreaded program does not protect access to the shared variable, the value of that variable may become corrupted. The sections of code that access shared data are often referred to as *critical sections*. To prevent multiple threads from changing data simultaneously, multithreaded programs typically restrict how many threads can execute the code in a critical section at a time. This restriction is accomplished through various *synchronization primitives*.

The **threading** module provides many thread synchronization primitives. The most primitive synchronization mechanism is the *lock*. A lock object (created with class **threading.Lock**) defines two methods—**acquire** and **release**. When a thread calls the **acquire** method, the lock enters its *locked* state. Once a lock has been acquired, no other threads may acquire the lock until the lock is released. This means that if another thread calls a lock's **acquire** method, the thread will block indefinitely. When the original thread calls the lock's **release** method, the lock enters the *unlocked* state and the blocked thread is *notified* (awakened). At this point, the previously blocked thread acquires the lock. If more than one thread is blocked on a lock, only one of those threads is notified.

Locks can be used to restrict access to a critical section. The program is written such that the thread must acquire a lock before entering a critical section and release the lock when exiting the critical section. Thus, if one thread is executing the critical section, any

other thread that attempts to enter the critical section will block until the original thread has exited the critical section.

Such a procedure provides only the most basic level of synchronization. Sometimes, however, we would like to create more sophisticated threads that access a critical section only when some event occurs (i.e., a data value has changed). This can be done by using *condition variables*. A thread uses a condition variable when the thread wants to monitor the state of some object or wants to be notified when some event occurs. When the object's state changes or the event occurs, blocked threads are notified. We discuss condition variables throughout this chapter in the context of the classic producer/consumer problem. The solution involves a consumer thread that accesses a critical section only when notified by a producer thread, and vice versa.

Condition variables are created with class **threading.Condition**. Because condition variables contain an *underlying lock*, condition variables provide **acquire** and **release** methods. Additional condition variable methods are **wait** and **notify**. When a thread has acquired the underlying lock, calling method **wait** releases the lock and causes the thread to block until it is awakened by a call to **notify** on the same condition variable. Calling method **notify** wakes up one thread waiting on the condition variable. All waiting threads can be woken up by invoking the condition variable's **notifyAll** method.

Semaphores (created with class **threading.Semaphore**) are synchronization primitives that allow a set number of threads to access a critical section. The **Semaphore** object uses a counter to keep track of the number of threads that acquire and release the semaphore. When a thread calls method **acquire**, the thread blocks if the counter is 0. Otherwise, the thread acquires the semaphore and method **acquire** decrements the counter. Calling method **release** releases the semaphore, increments the counter and notifies a waiting thread. The initial value of the internal counter can be passed as an argument to the **Semaphore** constructor (default is 1). Because the internal counter can never have a negative value, specifying a negative counter value raises an **AssertionError** exception.

Sometimes, one or more threads want to wait for a particular *event* to occur before proceeding with their execution. An **Event** object (created with class **threading.Event**) has an internal flag that is initially set to false (i.e., the event has not occurred). A thread that calls **Event** method **wait** blocks until the event occurs. When the event occurs, method **set** is called to set the flag to true and awaken all waiting threads. A thread that calls **wait** after the flag is true does not block at all. Method **isSet** returns true if the flag is true. Method **clear** sets the flag to false.

Writing a program that uses locks, condition variables or any other synchronization primitive takes careful scrutiny to ensure that the program does not *deadlock*. A program or thread deadlocks when the program or thread blocks forever on a needed resource. For example, consider the scenario where a thread enters a critical section that tries to open a file. If the file does not exists and the thread does not catch the exception, the thread terminates before releasing the lock. Now all other threads will deadlock, because they block indefinitely after they call the lock's **acquire** method.

Common Programming Error 19.1

Threads in the waiting state for a lock object must eventually be awakened explicitly (i.e., by releasing the lock) or the thread will wait forever. This may cause deadlock.

Testing and Debugging Tip 19.2

Be sure that every call to **`acquire`** *has a corresponding call to* **`release`** *that will eventually end the waiting.*

Performance Tip 19.4

Synchronization to achieve correctness in multithreaded programs can make programs run slower due to lock overhead and frequently moving threads between the running, waiting and ready states. There is not much to say, however, for highly efficient, incorrect multithreaded programs!

19.6 Producer/Consumer Relationship Without Thread Synchronization

In this section, we use a *producer/consumer relationship* to demonstrate the **`wait`** and **`notify`** methods of a condition variable. In a producer/consumer relationship, a *producer thread* calling a *produce* method may see that the *consumer thread* has not read the last message from a shared region of memory called a *buffer*, so the producer thread calls **`wait`** on a condition variable. When a consumer thread reads the message, it calls **`notify`** on the condition variable to allow a waiting producer to proceed. When a consumer thread calls a *consume* method and finds the buffer empty, it calls **`wait`**. A producer calling a produce method and finding the buffer empty, writes to the buffer, then calls **`notify`** so a waiting consumer can proceed.

Shared data can get corrupted if we do not synchronize access among multiple threads. Consider a producer/consumer relationship in which a producer thread deposits a sequence of numbers (we use 1, 2, 3, …) into a slot of shared memory. The consumer thread reads this data from the shared memory and prints the data. Figure 19.3 demonstrates a producer (defined in Fig. 19.4) and a consumer (defined in Fig. 19.5) accessing a single shared cell of memory without any synchronization (defined in Fig. 19.6). The program prints what the producer produces as it produces it and what the consumer consumes as it consumes it.

```
# Fig. 19.3: fig19_03.py
# Show multiple threads modifying shared object.

from UnsynchronizedInteger import UnsynchronizedInteger
from ProduceInteger import ProduceInteger
from ConsumeInteger import ConsumeInteger

# initialize integer and threads
number = UnsynchronizedInteger()
producer = ProduceInteger( "Producer", number )
consumer = ConsumeInteger( "Consumer", number )

print "Starting threads...\n"

# start threads
producer.start()
consumer.start()
```

Fig. 19.3 Threads modifying unsynchronized shared object (part 1 of 2).

```
18
19 # wait for threads to terminate
20 producer.join()
21 consumer.join()
22
23 print "\nAll threads have terminated."
```

```
Starting threads...

Producer setting sharedNumber to 1
Producer setting sharedNumber to 2
Consumer retrieving sharedNumber value 2
Consumer retrieving sharedNumber value 2
Consumer retrieving sharedNumber value 2
Producer setting sharedNumber to 3
Consumer retrieving sharedNumber value 3
Producer setting sharedNumber to 4
Consumer retrieving sharedNumber value 4
Consumer retrieving sharedNumber value 4
Consumer retrieving sharedNumber value 4
Producer setting sharedNumber to 5
Consumer retrieving sharedNumber value 5
Producer setting sharedNumber to 6
Producer setting sharedNumber to 7
Producer setting sharedNumber to 8
Producer setting sharedNumber to 9
Consumer retrieving sharedNumber value 9
Consumer retrieving sharedNumber value 9
Consumer retrieved values totaling: 44
Terminating Consumer
Producer setting sharedNumber to 10
Producer finished producing values
Terminating Producer

All threads have terminated.
```

Fig. 19.3 Threads modifying unsynchronized shared object (part 2 of 2).

Because the threads are not synchronized, data can be lost if the producer places new data into the slot before the consumer consumes the previous data, and data can be "doubled" if the consumer consumes data again before the producer produces the next item. To show these possibilities, the consumer thread in this example sums all the values it reads. The producer thread produces values from 1 to 10. If the consumer is able to read each value produced once, the sum would be 55. However, if you execute this program several times, you will see that the total is rarely, if ever, 55.

Figure 19.3 instantiates the shared **`UnsynchronizedInteger`** object **`number`** and uses it as the argument to the constructors for the **`ProduceInteger`** object **`producer`** and the **`ConsumeInteger`** object **`consumer`**. Next, the program invokes the **`Thread`** class **`start`** method on objects **`producer`** and **`consumer`** to place them in the *ready* state (lines 16–17). This statement launches the two threads. Lines 20–21 call **`Thread`** method **`join`** to ensure that the main program waits indefinitely for both threads

to terminate before continuing. Notice that line 23 is executed after both threads have terminated.

Class **ProduceInteger**—a subclass of **threading.Thread**—consists of attribute **sharedObject**, a constructor (lines 11–15) and a **run** method (lines 17–25). The constructor initializes attribute **sharedObject** to refer to the **UnsynchronizedInteger** object passed as an argument.

```
1  # Fig. 19.4: ProduceInteger.py
2  # Class that produces integers
3
4  import threading
5  import random
6  import time
7
8  class ProduceInteger( threading.Thread ):
9     """Thread to produce integers"""
10
11    def __init__( self, threadName, sharedObject ):
12       """Initialize thread, set shared object"""
13
14       threading.Thread.__init__( self, name = threadName )
15       self.sharedObject = sharedObject
16
17    def run( self ):
18       """Produce integers in range 1-10 at random intervals"""
19
20       for i in range( 1, 11 ):
21          time.sleep( random.randrange( 4 ) )
22          self.sharedObject.setSharedNumber( i )
23
24       print self.getName(), "finished producing values"
25       print "Terminating", self.getName()
```

Fig. 19.4 An integer-producer thread.

Class **ProduceInteger**'s **run** method consists of a **for** structure that loops ten times. Each iteration of the loop first invokes function **time.sleep** to put the **ProduceInteger** object into the *sleeping* state for a random time interval between 0 and 3 seconds. When the thread awakens, it invokes the shared object's **setSharedNumber** method (line 22) with the value of control variable **i** to set the shared object's data member. When the loop completes, the **ProduceInteger** thread displays a line in the command window indicating that it has finished producing data and terminates (i.e., the thread dies).

Class **ConsumeInteger**—a subclass of **threading.Thread**—consists of attribute **sharedObject**, a constructor (lines 11–15) and a **run** method (lines 17–29). The constructor initializes attribute **sharedObject** to refer to the **UnsynchronizedInteger** object passed as an argument.

```
1  # Fig. 19.5: ConsumeInteger.py
2  # Class that consumes integers
```

Fig. 19.5 An integer-consumer thread (part 1 of 2).

```
3
4  import threading
5  import random
6  import time
7
8  class ConsumeInteger( threading.Thread ):
9     """Thread to consume integers"""
10
11    def __init__( self, threadName, sharedObject ):
12       """Initialize thread, set shared object"""
13
14       threading.Thread.__init__( self, name = threadName )
15       self.sharedObject = sharedObject
16
17    def run( self ):
18       """Consume 10 values at random time intervals"""
19
20       sum = 0            # total sum of consumed values
21
22       # consume 10 values
23       for i in range( 10 ):
24          time.sleep( random.randrange( 4 ) )
25          sum += self.sharedObject.getSharedNumber()
26
27       print "%s retrieved values totaling: %d" % \
28          ( self.getName(), sum )
29       print "Terminating", self.getName()
```

Fig. 19.5 An integer-consumer thread (part 2 of 2).

Class **ConsumeInteger**'s **run** method consists of a **for** structure that loops ten times to read values from the **UnsynchronizedInteger** object to which **sharedObject** refers. Each iteration of the loop invokes function **time.sleep** to put the **ConsumeInteger** object into the *sleeping* state for a random time interval between 0 and 3 seconds. Next, the thread calls the **getSharedNumber** method to get the value of the shared object's data member. Then, the thread adds to variable **sum** the value returned by **getSharedInt** (line 25). When the loop completes, the **ConsumeInteger** thread displays a line in the command window indicating that it has finished consuming data and terminates (i.e., the thread dies).

Class **UnsynchronizedInteger**'s **setSharedNumber** method (lines 14–19) and **getSharedNumber** method (lines 21–28) do not synchronize access to instance variable **sharedNumber** (created in line 12). Ideally, we would like every value produced by the **ProduceInteger** object to be consumed exactly once by the **ConsumeInteger** object. However, the output of Fig. 19.3 reveals that the values 1, 6, 7, 8 and 10 are lost (i.e., never seen by the consumer) and the values 2, 4 and 9 are retrieved more than once by the consumer.

```
1  # Fig. 19.6: UnsynchronizedInteger.py
2  # Unsynchronized access to an integer
3
```

Fig. 19.6 Unsynchronized integer value class (part 1 of 2).

```
4  import threading
5
6  class UnsynchronizedInteger:
7     """Class that provides unsynchronized access an integer"""
8
9     def __init__( self ):
10       """Initialize shared number to -1"""
11
12       self.sharedNumber = -1
13
14    def setSharedNumber( self, newNumber ):
15       """Set value of integer"""
16
17       print "%s setting sharedNumber to %d" % \
18          ( threading.currentThread().getName(), newNumber )
19       self.sharedNumber = newNumber
20
21    def getSharedNumber( self ):
22       """Get value of integer"""
23
24       tempNumber = self.sharedNumber
25       print "%s retrieving sharedNumber value %d" % \
26          ( threading.currentThread().getName(), tempNumber )
27
28       return tempNumber
```

Fig. 19.6 Unsynchronized integer value class (part 2 of 2).

In fact, method **getSharedNumber** must perform some "tricks" to make the output accurately reflect the value of the data member. Line 24 assigns the value of data member **sharedNumber** to variable **tempNumber**. Lines 25–28 then use the value of **tempNumber** to print the message and return the value. If we did not use a temporary variable in this way, the following scenario could occur. The consumer could call method **getSharedNumber** and print a message that displays the value of the data member. The interpreter might then switch out the consumer thread for the producer thread. The producer thread might then change the value of **sharedNumber** any number of times by calling method **setSharedNumber**. Eventually, the interpreter switches the consumer back in and method **getSharedNumber** returns a value different that the value printed before the consumer was switched out.

This example clearly demonstrates that access to shared data by concurrent threads must be controlled carefully or a program may produce incorrect results. To solve the problems of lost data and doubled data in the previous example, we must synchronize access to the shared data for the concurrent producer and consumer threads.

19.7 Producer/Consumer Relationship With Thread Synchronization

The program in Fig. 19.7 demonstrates a producer and a consumer accessing a shared cell of memory with synchronization, so that the consumer consumes exactly one time after the producer produces each value. The program differs only in that it passes an object of class

SynchronizedInteger to the producer and consumer. Classes **ProduceInteger** and **ConsumeInteger** are identical to the previous section.

```
# Fig. 19.7: fig19_07.py
# Show multiple threads modifying shared object.

from SynchronizedInteger import SynchronizedInteger
from ProduceInteger import ProduceInteger
from ConsumeInteger import ConsumeInteger

# initialize number and threads
number = SynchronizedInteger()
producer = ProduceInteger( "Producer", number )
consumer = ConsumeInteger( "Consumer", number )

print "Starting threads...\n"

# start threads
producer.start()
consumer.start()

# wait for threads to terminate
producer.join()
consumer.join()

print "\nAll threads have terminated."
```

Fig. 19.7 Threads modifying a synchronized shared object (part 1 of 2).

```
Starting threads...

Producer setting sharedNumber to 1
Consumer retrieving sharedNumber value 1
Producer setting sharedNumber to 2
Consumer retrieving sharedNumber value 2
Producer setting sharedNumber to 3
Consumer retrieving sharedNumber value 3
Producer setting sharedNumber to 4
Consumer retrieving sharedNumber value 4
Producer setting sharedNumber to 5
Consumer retrieving sharedNumber value 5
Producer setting sharedNumber to 6
Consumer retrieving sharedNumber value 6
Producer setting sharedNumber to 7
Consumer retrieving sharedNumber value 7
Producer setting sharedNumber to 8
Consumer retrieving sharedNumber value 8
Producer setting sharedNumber to 9
Consumer retrieving sharedNumber value 9
Producer setting sharedNumber to 10
Producer finished producing values
Terminating Producer
Consumer retrieving sharedNumber value 10
Consumer retrieved values totaling: 55
Terminating Consumer

All threads have terminated.
```

Fig. 19.7 Threads modifying a synchronized shared object (part 2 of 2).

Class **SynchronizedInteger** (Fig. 19.8) contains three attributes—**sharedNumber**, **writeable** and **threadCondition**, a condition variable. Method **setSharedNumber** uses the condition variable to determine if the thread that calls the method can write to the shared memory location. Method **getSharedNumber** uses the condition variable to determine if the calling thread can read from the shared memory location. Line 14 creates the thread condition variable by invoking the **threading.Condition** constructor. Because no argument (i.e., an underlying lock) is passed to the condition variable's constructor, a new lock will be created for the condition variable.

```
1  # Fig. 19.8: SynchronizedInteger.py
2  # Synchronized access to an integer with condition variable
3
4  import threading
5
6  class SynchronizedInteger:
7     """Class that provides synchronized access an integer"""
8
9     def __init__( self ):
10       """Set shared number, write flag and condition variable"""
```

Fig. 19.8 Synchronized integer value class (part 1 of 2).

```
11
12       self.sharedNumber = -1
13       self.writeable = 1      # the value can be changed
14       self.threadCondition = threading.Condition()
15
16    def setSharedNumber( self, newNumber ):
17       """Set value of integer--blocks until lock acquired"""
18
19       # block until lock released then acquire lock
20       self.threadCondition.acquire()
21
22       # while not producer's turn, release lock and block
23       while not self.writeable:
24          self.threadCondition.wait()
25
26       # (lock has now been re-acquired)
27
28       print "%s setting sharedNumber to %d" % \
29          ( threading.currentThread().getName(), newNumber )
30       self.sharedNumber = newNumber
31
32       self.writeable = 0              # allow consumer to consume
33       self.threadCondition.notify()   # wake up a waiting thread
34       self.threadCondition.release()  # allow lock to be acquired
35
36    def getSharedNumber( self ):
37       """Get value of integer--blocks until lock acquired"""
38
39       # block until lock released then acquire lock
40       self.threadCondition.acquire()
41
42       # while producer's turn, release lock and block
43       while self.writeable:
44          self.threadCondition.wait()
45
46       # (lock has now been re-acquired)
47
48       tempNumber = self.sharedNumber
49       print "%s retrieving sharedNumber value %d" % \
50          ( threading.currentThread().getName(), tempNumber )
51
52       self.writeable = 1              # allow producer to produce
53       self.threadCondition.notify()   # wake up a waiting thread
54       self.threadCondition.release()  # allow lock to be acquired
55
56       return tempNumber
```

Fig. 19.8 Synchronized integer value class (part 2 of 2).

The constructor (lines 9–14) creates attribute **writeable** and initializes its value to 1. The class' condition variable—**threadCondition**—protects access to attribute **writeable**. If **writeable** is 1, a producer can place a value into variable **sharedNumber**. However, this means that a consumer currently cannot read the value of **sharedNumber**. If **writeable** is 0, a consumer can read a value from variable

sharedNumber. However, this means that a producer currently cannot place a value into **sharedNumber**.

When the **ProduceInteger** thread object invokes method **setSharedNumber** (lines 16–34), a lock is **acquire**d on the condition variable (line 20). The **while** structure in lines 23–24 tests the **writeable** data member. If **writeable** is 0, line 24 invokes the condition variable's **wait** method. This call places the **ProduceInteger** thread object that called method **setSharedNumber** into the *waiting* state and releases the lock on the **SynchronizedInteger** object so other objects may access it.

The **ProduceInteger** object remains in the *waiting* state until it is notified that it may proceed—at which point it enters the *ready* state and waits for the interpreter to execute it. When the **ProduceInteger** object reenters the *running* state, the object implicitly reacquires the lock on the condition variable, and the **setSharedNumber** method continues executing in the **while** structure with the next statement after **wait**. There are no more statements, so the **while** condition is tested again. If the condition is true (i.e., **writeable** is 0), the program displays a message indicating that the producer is setting **sharedNumber** to a new value, **newNumber** (the argument passed to **setSharedNumber**). **writeable** is set to 0 to indicate that the shared memory is now full (i.e., a consumer can read the value and a producer cannot put another value there yet) and condition variable method **notify** is invoked. If there are any waiting threads, one thread in the *waiting* state is placed into the *ready* state, indicating that the thread can now attempt its task again (as soon as it is switched into the interpreter). Lines 34 then calls condition variable method **release**, and method **setSharedNumber** returns to its caller.

Common Programming Error 19.2

*Condition variable method **notify** does not release the underlying lock. Forgetting to call **release** can result in deadlock.*

Methods **getSharedNumber** and **setSharedNumber** are implemented similarly. When the **ConsumeInteger** object invokes method **getSharedNumber**, the method acquires a lock on the condition variable object. The **while** structure in lines 43–44 tests variable **writeable**. If **writeable** is 1 (i.e., there is nothing to consume), the condition variable's **wait** method is invoked. This places the **ConsumeInteger** thread object that called method **getSharedNumber** into the *waiting* state and releases the lock on the **SynchronizedInteger** object so other objects may access it. The **ConsumeInteger** object remains in the *waiting* state until it is notified that it may proceed—at which point it enters the *ready* state and waits for the interpreter to switch it in. When the **ConsumeInteger** object reenters the *running* state, the **setSharedNumber** method reacquires the lock on the condition variable object and the method continues executing in the **while** structure with the next statement after **wait**. There are no more statements, so the **while** condition is tested again. If the condition is 0, the value of **sharedNumber** is stored in variable **tempNumber** (line 48) and the method outputs a message to the command window indicating that the consumer is retrieving **sharedNumber**. Note that the value of **sharedNumber** is only retrieved once and stored in variable **tempNumber** (while within the critical section). Lines 49 and 56 (outside the critical section) use the value of **tempNumber** rather than **sharedNumber** to ensure that they use the same value.

Next, **writeable** is set to 1 to indicate that the shared memory is now empty, and condition variable method **notify** is invoked. If there are any waiting threads, one thread in the *waiting* state is placed into the *ready* state, indicating that the thread can now attempt its task again (as soon as it is assigned a processor). Line 54 releases the lock on the condition variable, and line 56 returns the value of **tempNumber** to **getSharedNumber**'s caller.

The output in Fig. 19.7 shows that every integer produced is consumed once—no values are lost and no values are doubled. Also, the consumer cannot read a value until the producer produces a value. The next section addresses a way for consumers and producers to read and write multiple values simultaneously.

19.8 Producer/Consumer Relationship: The Circular Buffer

The program of Fig. 19.7 does access the shared data correctly, but it may not perform optimally. Because the threads are running asynchronously, we cannot predict their relative speeds. If the producer wants to produce faster than the consumer can consume, it cannot do so. To enable the producer to continue producing we can use a *circular buffer* which has enough cells to handle the "extra" production. The program of Fig. 19.9 demonstrates a producer and a consumer accessing a synchronized circular buffer (in this case, a shared list of five cells). Consumer only consumes a value when the list contains one or more values; the producer only produces a value when the list contains one or more available cells.

```
# Fig. 19.9: fig19_09.py
# Show multiple threads modifying shared object.

from SynchronizedCells import SynchronizedCells
from ProduceInteger import ProduceInteger
from ConsumeInteger import ConsumeInteger

# initialize number and threads
number = SynchronizedCells()
producer = ProduceInteger( "Producer", number )
consumer = ConsumeInteger( "Consumer", number )

print "Starting threads...\n"

# start threads
producer.start()
consumer.start()

# wait for threads to terminate
producer.join()
consumer.join()

print "\nAll threads have terminated."
```

Fig. 19.9 Threads modifying a synchronized circular buffer (part 1 of 2).

```
Starting threads...

WAITING TO CONSUME
Produced  1 into cell 0    write 1 read 0   [1, -1, -1, -1, -1]
Consumed  1 from cell 0    write 1 read 1   [-1, -1, -1, -1, -1]
BUFFER EMPTY
Produced  2 into cell 1    write 2 read 1   [-1, 2, -1, -1, -1]
Produced  3 into cell 2    write 3 read 1   [-1, 2, 3, -1, -1]
Produced  4 into cell 3    write 4 read 1   [-1, 2, 3, 4, -1]
Consumed  2 from cell 1    write 4 read 2   [-1, -1, 3, 4, -1]
Produced  5 into cell 4    write 0 read 2   [-1, -1, 3, 4, 5]
Produced  6 into cell 0    write 1 read 2   [6, -1, 3, 4, 5]
Produced  7 into cell 1    write 2 read 2   [6, 7, 3, 4, 5]
BUFFER FULL
WAITING TO PRODUCE 8
Consumed  3 from cell 2    write 2 read 3   [6, 7, -1, 4, 5]
Produced  8 into cell 2    write 3 read 3   [6, 7, 8, 4, 5]
BUFFER FULL
Consumed  4 from cell 3    write 3 read 4   [6, 7, 8, -1, 5]
Produced  9 into cell 3    write 4 read 4   [6, 7, 8, 9, 5]
BUFFER FULL
WAITING TO PRODUCE 10
Consumed  5 from cell 4    write 4 read 0   [6, 7, 8, 9, -1]
Produced 10 into cell 4    write 0 read 0   [6, 7, 8, 9, 10]
BUFFER FULL
Producer finished producing values
Terminating Producer
Consumed  6 from cell 0    write 0 read 1   [-1, 7, 8, 9, 10]
Consumed  7 from cell 1    write 0 read 2   [-1, -1, 8, 9, 10]
Consumed  8 from cell 2    write 0 read 3   [-1, -1, -1, 9, 10]
Consumed  9 from cell 3    write 0 read 4   [-1, -1, -1, -1, 10]
Consumed 10 from cell 4    write 0 read 0   [-1, -1, -1, -1, -1]
BUFFER EMPTY
Consumer retrieved values totaling: 55
Terminating Consumer

All threads have terminated.
```

Fig. 19.9 Threads modifying a synchronized circular buffer (part 2 of 2).

Class **SynchronizedCells** (Fig. 19.10) contains six attributes—**sharedCells** is a five-element list of integers that represents the circular buffer, **writeable** indicates whether a producer can write into the circular buffer, **readable** indicates whether a consumer can read from the circular buffer, **readLocation** indicates the current position from which the next value can be read by a consumer, **writeLocation** indicates the next location in which a value can be placed by a producer and **threadCondition** is the condition variable that protects access to the buffer.

```
# Fig. 19.10: SynchronizedCells.py
# Synchronized circular buffer of integer values
```

Fig. 19.10 Synchronized circular buffer of integers (part 1 of 3).

```

import threading

class SynchronizedCells:

   def __init__( self ):
      """Set cells, flags, locations and condition variable"""

      self.sharedCells = [ -1, -1, -1, -1, -1 ]  # buffer
      self.writeable = 1         # buffer may be changed
      self.readable = 0          # buffer may not be read
      self.writeLocation = 0     # current writing index
      self.readLocation = 0      # current reading index

      self.threadCondition = threading.Condition()

   def setSharedNumber( self, newNumber ):
      """Set next buffer index value--blocks until lock acquired"""

      # block until lock released then acquire lock
      self.threadCondition.acquire()

      # while buffer is full, release lock and block
      while not self.writeable:
         print "WAITING TO PRODUCE", newNumber
         self.threadCondition.wait()

      # buffer is not full, lock has been re-acquired

      # produce a number in shared cells, consumer may consume
      self.sharedCells[ self.writeLocation ] = newNumber
      self.readable = 1
      print "Produced %2d into cell %d" % \
         ( newNumber, self.writeLocation ),

      # set writing index to next place in buffer
      self.writeLocation = ( self.writeLocation + 1 ) % 5

      print "   write %d read %d   " % \
         ( self.writeLocation, self.readLocation ),
      print self.sharedCells

      # if producer has caught up to consumer, buffer is full
      if self.writeLocation == self.readLocation:
         self.writeable = 0
         print "BUFFER FULL"

      self.threadCondition.notify()  # wake up a waiting thread
      self.threadCondition.release() # allow lock to be acquired

   def getSharedNumber( self ):
      """Get next buffer index value--blocks until lock acquired"""

      # block until lock released then acquire lock
```

Fig. 19.10 Synchronized circular buffer of integers (part 2 of 3).

```
57        self.threadCondition.acquire()
58
59        # while buffer is empty, release lock and block
60        while not self.readable:
61           print "WAITING TO CONSUME"
62           self.threadCondition.wait()
63
64        # buffer is not empty, lock has been re-acquired
65
66        # consume a number from shared cells, producer may produce
67        self.writeable = 1
68        tempNumber = self.sharedCells[ self.readLocation ]
69        self.sharedCells[ self.readLocation ] = -1
70
71        print "Consumed %2d from cell %d" % \
72           ( tempNumber, self.readLocation ),
73
74        # move to next produced number
75        self.readLocation = ( self.readLocation + 1 ) % 5
76
77        print "   write %d read %d  " % \
78           ( self.writeLocation, self.readLocation ),
79        print self.sharedCells
80
81        # if consumer has caught up to producer, buffer is empty
82        if self.readLocation == self.writeLocation:
83           self.readable = 0
84           print "BUFFER EMPTY"
85
86        self.threadCondition.notify()  # wake up a waiting thread
87        self.threadCondition.release() # allow lock to be acquired
88
89        return tempNumber
```

Fig. 19.10 Synchronized circular buffer of integers (part 3 of 3).

Method **setSharedNumber** (lines 19–51) performs the same tasks as it did in Fig. 19.8 with a few modifications. When execution continues at line 33 after the **while** loop, the produced value is placed into the circular buffer at location **writeLocation**. Next, **readable** is set to 1 because there is at least one value in the buffer to be read. The method prints the produced value and the cell in which the value was placed. Then, **writeLocation** is updated for the next call to **setSharedNumber**. Note that the value of **writeLocation** is kept in range 0–4, inclusive, using the **%** operator. The output is continued with the current **writeLocation** and **readLocation** values and the values in the circular buffer. If the **writeLocation** is equal to the **readLocation**, the circular buffer currently is full, so **writeable** is set to 0 and the string **"BUFFER FULL"** is displayed. Next, condition variable method **notify** is invoked to indicate that a waiting thread should move to the *ready* state. Finally, condition variable method **release** is invoked to release the condition variable's underlying lock.

Method **getSharedNumber** (line 53–89) also performs the same tasks in this example as it did in Fig. 19.8 with a few modifications. When execution continues at line 67 after the **while** loop, **writeable** is set to 1 because there is at least one open position

in the buffer in which a value can be placed. Next, the method assigns to **tempNumber** the value at location **readLocation** in the circular buffer. Line 69 sets the value at location **readLocation** in the buffer to –1, indicating it is an empty spot. The value consumed and the cell from which the value was read are printed. Then, the method updates attribute **readLocation** for the next call to method **getSharedNumber**. The output continues with the current **writeLocation** and **readLocation** values and the current values in the circular buffer. If the **readLocation** is equal to the **writeLocation**, the circular buffer is currently empty, so **readable** is set to 0 and the string **"BUFFER EMPTY"** is displayed. Next, line 86 invokes condition variable method **notify** to place the next waiting thread into the *ready* state. Line 87 invokes condition variable method **release** to release the condition variable's underlying lock. Finally, line 89 returns the retrieved value to the calling thread.

We have modified the program of Fig. 19.9 to include the current **writeLocation** and **readLocation** values. We also display the current contents of the buffer **sharedCells**. The elements of the **sharedCells** list were initialized to –1 for output purposes so you can see each value inserted into the buffer. Notice that after the fifth value is placed in the fifth element of the buffer, the sixth value is inserted at the beginning of the list—thus providing the *circular buffer* effect.

19.9 Semaphores

A *semaphore* is a variable that controls access to a common resource or a critical section. A semaphore maintains a counter that specifies the number of threads that can use the resource or enter the critical section. The counter is decremented each time a thread acquires the semaphore. When the counter is zero, the semaphore blocks any other threads until the semaphore has been released by another thread. Figure 19.11 uses a restaurant scenario to demonstrate using semaphores to control access to a critical section.

```
# Figure 19.11: fig19_11.py
# Using a semaphore to control access to a critical section

import threading
import random
import time

class SemaphoreThread( threading.Thread ):
   """Class using semaphores"""

   availableTables = [ "A", "B", "C", "D", "E" ]

   def __init__( self, threadName, semaphore ):
      """Initialize thread"""

      threading.Thread.__init__( self, name = threadName )
      self.sleepTime = random.randrange( 1, 6 )

      # set the semaphore as a data attribute of the class
      self.threadSemaphore = semaphore
```

Fig. 19.11 Using a semaphore to control access to a critical section (part 1 of 3).

```
    def run( self ):
        """Print message and release semaphore"""

        # acquire the semaphore
        self.threadSemaphore.acquire()

        # remove a table from the list
        table = SemaphoreThread.availableTables.pop()
        print "%s entered; seated at table %s."  % \
           ( self.getName(), table ),
        print SemaphoreThread.availableTables

        time.sleep( self.sleepTime )   # enjoy a meal

        # free a table
        print "   %s exiting; freeing table %s." % \
           ( self.getName(), table ),
        SemaphoreThread.availableTables.append( table )
        print SemaphoreThread.availableTables

        # release the semaphore after execution finishes
        self.threadSemaphore.release()

threads = []  # list of threads

# semaphore allows five threads to enter critical section
threadSemaphore = threading.Semaphore(
   len( SemaphoreThread.availableTables ) )

# create ten threads
for i in range( 1, 11 ):
   threads.append( SemaphoreThread( "thread" + str( i ),
      threadSemaphore ) )

# start each thread
for thread in threads:
   thread.start()
```

Fig. 19.11 Using a semaphore to control access to a critical section (part 2 of 3).

```
thread1 entered; seated at table E. ['A', 'B', 'C', 'D']
thread2 entered; seated at table D. ['A', 'B', 'C']
thread3 entered; seated at table C. ['A', 'B']
thread4 entered; seated at table B. ['A']
thread5 entered; seated at table A. []
   thread2 exiting; freeing table D. ['D']
thread6 entered; seated at table D. []
   thread1 exiting; freeing table E. ['E']
thread7 entered; seated at table E. []
   thread3 exiting; freeing table C. ['C']
thread8 entered; seated at table C. []
   thread4 exiting; freeing table B. ['B']
thread9 entered; seated at table B. []
   thread5 exiting; freeing table A. ['A']
thread10 entered; seated at table A. []
   thread7 exiting; freeing table E. ['E']
   thread8 exiting; freeing table C. ['E', 'C']
   thread9 exiting; freeing table B. ['E', 'C', 'B']
   thread10 exiting; freeing table A. ['E', 'C', 'B', 'A']
   thread6 exiting; freeing table D. ['E', 'C', 'B', 'A', 'D']
```

Fig. 19.11 Using a semaphore to control access to a critical section (part 3 of 3).

Lines 48–49 create a **threading.Semaphore** instance that allows five threads to access the critical section at a time. Lines 52–54 create a list of **SemaphoreThread** instances. Method **start** starts each thread in the list (lines 57–58) .

Class **SemaphoreThread** (lines 8–43) represents a single customer at a restaurant. Class attribute **availableTables** (line 11) keeps track of the available tables in the restaurant.

A semaphore has a built-in counter to keep track of the number of calls to its **acquire** and **release** methods. If the counter is greater than zero, method **acquire** (line 26) obtains the semaphore for the thread and decrements the counter. If the counter is zero, the thread blocks until another thread releases the semaphore.

List method **pop** (line 29) removes the last item from **availableTables** as another thread begins executing the critical section. The program displays which thread entered the critical section and the thread sleeps for a randomly determined time. Line 39 appends the removed item to **availableTables** as a thread prepares to exit the critical section.

Semaphore method **release** (line 43) releases the semaphore when the thread finishes executing the critical section. The method call increments the counter and notifies a waiting thread.

Note that if lines 26 and 43 are removed from Fig. 19.11, more than five threads may attempt to remove an item from the shared list, resulting in an **IndexError** exception.

19.10 Events

Module **threading** defines class **Event**, which is useful for thread communication. An **Event** object has an internal flag, which is either true or false. One or more threads may

call the **Event** object's **wait** method to block until the event occurs. When the event occurs, the blocked thread or threads are notified and resume execution. Figure 19.12 illustrates a situation where a traffic light turns green every 3 seconds.

```
# Fig. 19.12: fig19_12.py
# Demonstrating Event objects

import threading
import random
import time

class VehicleThread( threading.Thread ):
   """Class representing a motor vehicle at an intersection"""

   def __init__( self, threadName, event ):
      """Initializes thread"""

      threading.Thread.__init__( self, name = threadName )

      # ensures that each vehicle waits for a green light
      self.threadEvent = event

   def run( self ):
      """Vehicle waits unless/until light is green"""

      # stagger arrival times
      time.sleep( random.randrange( 1, 10 ) )

      # prints arrival time of car at intersection
      print "%s arrived at %s" % \
         ( self.getName(), time.ctime( time.time() ) )

      # flag is false until two vehicles are queued
      self.threadEvent.wait()

      # displays time that car departs intersection
      print "%s passes through intersection at %s" % \
         ( self.getName(), time.ctime( time.time() ) )

greenLight = threading.Event()
vehicleThreads = []

# creates and starts ten Vehicle threads
for i in range( 1, 11 ):
   vehicleThreads.append( VehicleThread( "Vehicle" + str( i ),
      greenLight ) )

for vehicle in vehicleThreads:
   vehicle.start()

while threading.activeCount() > 1:

   # sets the Event object's flag to false
```

Fig. 19.12 Traffic light example demonstrating an **Event** object (part 1 of 2).

```
50      greenLight.clear()
51      print "RED LIGHT!"
52
53      time.sleep( 3 )
54
55      # sets the Event object's flag to true
56      print "GREEN LIGHT!"
57      greenLight.set()
```

```
RED LIGHT!
Vehicle4 arrived at Mon Aug 20 16:58:33 2001
Vehicle8 arrived at Mon Aug 20 16:58:33 2001
Vehicle9 arrived at Mon Aug 20 16:58:35 2001
Vehicle10 arrived at Mon Aug 20 16:58:35 2001
GREEN LIGHT!
Vehicle4 passes through intersection at Mon Aug 20 16:58:35 2001
Vehicle8 passes through intersection at Mon Aug 20 16:58:35 2001
Vehicle9 passes through intersection at Mon Aug 20 16:58:35 2001
Vehicle10 passes through intersection at Mon Aug 20 16:58:35 2001
RED LIGHT!
Vehicle2 arrived at Mon Aug 20 16:58:36 2001
Vehicle5 arrived at Mon Aug 20 16:58:37 2001
Vehicle7 arrived at Mon Aug 20 16:58:37 2001
GREEN LIGHT!
Vehicle2 passes through intersection at Mon Aug 20 16:58:38 2001
Vehicle5 passes through intersection at Mon Aug 20 16:58:38 2001
Vehicle7 passes through intersection at Mon Aug 20 16:58:38 2001
RED LIGHT!
Vehicle1 arrived at Mon Aug 20 16:58:39 2001
Vehicle6 arrived at Mon Aug 20 16:58:40 2001
Vehicle3 arrived at Mon Aug 20 16:58:41 2001
GREEN LIGHT!
Vehicle1 passes through intersection at Mon Aug 20 16:58:41 2001
Vehicle6 passes through intersection at Mon Aug 20 16:58:41 2001
Vehicle3 passes through intersection at Mon Aug 20 16:58:41 2001
```

Fig. 19.12 Traffic light example demonstrating an **Event** object (part 2 of 2).

Line 36 creates an **Event** instance—**greenLight**—which simulates a traffic light. Lines 40–42 create a list of **VehicleThread**s. Class **VehicleThread** (lines 8–34) represents a vehicle at the intersection as a thread. Lines 45–46 start each vehicle thread. Each thread sleeps for a random amount of time, prints an arrival message, waits until the traffic light is green (i.e., **greenLight**'s internal flag is true) and prints a departing message.

The **while** structure in lines 48–58 loops until only the main thread is left (i.e., all vehicle threads have terminated). Each iteration calls **Event** method **clear**, sleeps for 3 seconds and calls **Event** method **set**. **Event** methods **clear** and **set** change the value of an internal flag to false and true, respectively.

19.11 Daemon Threads

A *daemon thread* is a thread that runs for the benefit of other threads. Daemon threads run in the background (i.e., when processor time is available that would otherwise go to waste). Unlike conventional user threads, daemon threads do not prevent a program from terminating. The garbage collector is a daemon thread. Non-daemon threads are conventional user threads. We designate a thread as a daemon with the method call

```
setDaemon( 1 )
```

An argument of 0 means that the thread is not a daemon thread. A program can include a mixture of daemon threads and non-daemon threads. When only daemon threads remain in a program, the program exits. If a thread is to be a daemon, it must be set as such before its **`start`** method is called; otherwise, **`setDaemon`** raises an **`AssertionError`** exception. Method **`isDaemon`** returns 1 if a thread is a daemon thread and returns 0 otherwise.

SUMMARY

[*To be done for second round of review***]**

TERMINOLOGY

[*To be done for second round of review***]**

SELF-REVIEW EXERCISES

[*To be done for second round of review***]**

ANSWERS TO SELF REVIEW EXERCISES

[*To be done for second round of review***]**

EXERCISES

[*To be done for second round of review***]**

Notes to Reviewers:

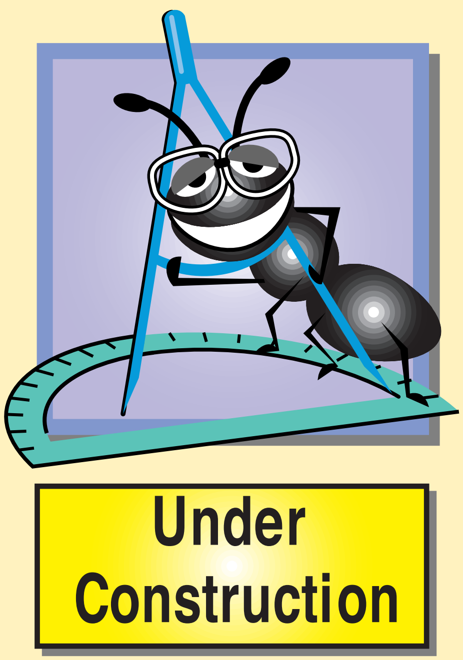
- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **ben.wiedermann@deitel.net**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copy edited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are mostly concerned with technical correctness and correct use of idiom. We will not make significant adjustments to our writing or coding style on a global scale. Please send us a short e-mail if you would like to make a suggestion.
- If you find something incorrect, please show us how to correct it.
- In the later round(s) of review, please read all the back matter, including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

20

Networking

Objectives

- To understand the elements of Python networking with URLs, sockets and datagrams.
- To implement Python networking applications using sockets and datagrams.
- To understand how to implement Python clients and servers that communicate with one another.
- To understand how to implement network-based collaborative applications.
- To construct a multithreaded server.



If the presence of electricity can be made visible in any part of a circuit, I see no reason why intelligence may not be transmitted instantaneously by electricity.
Samuel F. B. Morse

Mr. Watson, come here, I want you.
Alexander Graham Bell

What networks of railroads, highways and canals were in another age, the networks of telecommunications, information and computerization … are today.
Bruno Kreisky, Austrian Chancellor

Science may never come up with a better office-communication system than the coffee break.
Earl Wilson

It's currently a problem of access to gigabits through punybaud.
J. C. R. Licklider

Outline

20.1 Introduction

In this chapter, our discussion focuses on several fundamental networking technologies that can be used to build distributed applications. We revisit the client/server relationship between World Wide Web browsers and World Wide Web servers to demonstrate a script that causes the Web browser to load a new Web page.

Because Python is such a high-level language, networking tasks that take a great deal of code and effort in other languages can be accomplished easily and simply in Python. This chapter highlights the most frequently used Python networking capabilities. We demonstrate module **`urllib`** and its ability to obtain a document downloaded from the World Wide Web.

We also introduce Python's *socket-based communications*, which enable applications to view networking as if it were file I/O—a program can receive from a *socket* or send to a socket as simply as reading from a file or writing to a file. We show how to create and manipulate sockets.

Python provides *stream sockets* and *datagram sockets***.** With *stream sockets* a process establishes a *connection* to another process. While the connection is in place, data flows between the processes in continuous *streams*. Stream sockets are said to provide a *connection-oriented service*. The protocol used for transmission is the popular *TCP (Transmission Control Protocol)*.

With *datagram sockets*, individual *packets* of information are transmitted. This is not the right protocol for everyday users because unlike TCP, the protocol used, *UDP*—the *User Datagram Protocol*, is a *connectionless service*, and does not guarantee that packets arrive in any particular order. In fact, packets can be lost, can be duplicated and can even arrive out of sequence. So with UDP, significant extra programming is required on the user's part to deal with these problems (if the user chooses to do so). Stream sockets and the TCP protocol will be the most desirable for the vast majority of Python programmers.

Performance Tip 20.1

Connectionless services generally offer greater performance but less reliability than connection-oriented services.

Portability Tip 20.1

The TCP protocol and its related set of protocols enable a great variety of heterogeneous computer systems (i.e., computer systems with different processors and different operating systems) to intercommunicate.

Once again, we will see that many of the networking details for the examples in this chapter are handled by the Python modules we use.

20.2 Accessing URLs over HTTP

The Internet offers many protocols. The **`http`** protocol (HyperText Transfer Protocol) that forms the basis of the World Wide Web uses URLs (Uniform Resource Locators, also called Universal Resource Locators) to locate data on the Internet. Common URLs represent files or directories and can represent complex tasks such as database lookups and Internet searches. If you know the URL of publicly available XHTML files anywhere on the World Wide Web, you can access that data through **`http`**.

Figure 20.1 uses **`Tkinter`** and **`Pmw`** GUI components to display the contents of a file on a Web server. We define class **`WebBrowser`** that acts as a simple Web browser. The user inputs the URL in the **`Entry`** at the top of the window and the corresponding Web document (if it exists) is displayed in the **`ScrolledText`**.

```
# Fig. 20.1: fig20_01.py
# This program displays the contents of a file on a Web server.

from Tkinter import *
import Pmw
import urllib
import urlparse

class WebBrowser( Frame ):
   "A simple Web browser"

   def __init__( self ):
      "Create the Web browser GUI"

      Frame.__init__( self )
      Pmw.initialise()
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Simple Web Browser" )
      self.master.geometry( "400x300" )

      self.address = Entry( self )
      self.address.pack( fill = X, padx = 5, pady = 5 )
      self.address.bind( "<Return>", self.getPage )

      self.contents = Pmw.ScrolledText( self,
         text_state = DISABLED )
      self.contents.pack( expand = YES, fill = BOTH, padx = 5,
         pady = 5 )
```

Fig. 20.1 Reading a file through a URL connection (part 1 of 3).

```
   def getPage( self, event ):
      "Parse the URL, add addressing scheme and retrieve file"

      # parse the URL
      myURL = event.widget.get()
      components = urlparse.urlparse( myURL )
      self.contents.text_state = NORMAL

      # if addressing scheme not specified, use http
      if components[ 0 ] == "":
         myURL = "http://" + myURL

      # connect and retrieve the file
      try:
         tempFile = urllib.urlopen( myURL )
         self.contents.settext( tempFile.read() ) # show results
      except IOError:
         self.contents.settext( "Error finding file" )

      self.contents.text_state = DISABLED

def main():
   WebBrowser().mainloop()

if __name__ == "__main__":
   main()
```

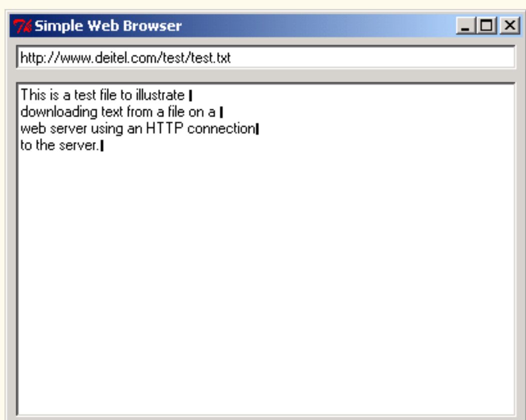


Fig. 20.1 Reading a file through a URL connection (part 2 of 3).



Fig. 20.1 Reading a file through a URL connection (part 3 of 3).

Class **WebBrowser** contains an **Entry** component **address**, in which the user enters the URL of the file to read, and **ScrolledText** component **contents** that displays the contents of the file. When the user presses the *Enter* key in the **Entry** component, method **getPage** executes. Method **getPage** (lines 30–49) retrieves the specified file from the Web server. Line 34 obtains the URL from component **address** by invoking its **get** method.

Module **urlparse**, a module that facilitates the manipulation URLs, parses the URL. Function **urlparse.urlparse** takes a string as input and returns a six-element tuple. The first element of the tuple is known as the *addressing scheme*. This example uses **http** as the addressing scheme. The World Wide Web uses HyperText Transfer Protocol (HTTP) to define how Web servers and browsers respond to commands. Entering a URL beginning with **http** directs the Web server to retrieve and transfer the requested URL document. Line 39 checks if the user has entered a URL beginning with **"http://"**. If not, the program, assuming that the user has simply forgotten it, adds it to the URL (line 40).

Lines 43–47 attempt to connect to the Web server and retrieve the file using module **urllib**. Module **urllib** provides methods for accessing data over the Internet. Line 44 passes the URL to **urllib** function **urlopen** to retrieve the file. The function performs a *DNS* (*Domain Name System* or *Service*) lookup. DNS translates a domain name, or URL, into an *IP address*, a unique identifier for a computer on a network. The module searches the Web server for the requested document. If successful, **urlopen** returns a Python file object. Line 45 reads the file and displays the results in the component **contents**. If **urlopen** fails, line 47 displays a message to the user.

20.3 Establishing a Simple Server (Using Stream Sockets)

Module **socket** contains the function and class definitions that provide the capabilities to build programs that communicate with one another over a network. Establishing a simple server in Python requires six steps. Step 1 is to create a ***socket*** object. A call to the **socket** constructor

```
socket = socket.socket( family, type )
```

creates a new socket using the specified address family and type. Argument *family* can be either **AF_INET** or **AF_UNIX**. In this chapter, we use only **AF_INET**. The most common values for argument *type* are ***SOCK_STREAM*** (for stream sockets) and ***SOCK_DGRAM*** (for datagram sockets). Note that these constants are defined in module **socket**. For the purposes of our discussion, we assume that we have created a stream socket. Section 20.6 discusses datagram sockets.

Once a **socket** is created, it must be bound to an address (step 2). A call to a **socket** instance's **bind** method such as

```
socket.bind( address )
```

binds the socket to the specified *address*. For a socket created by specifying family **AF_INET**, *address* must be a two-element tuple in the form (*host*, *port*), where *host* is a string representing the remote machine's hostname or an IP address, and *port* is a port number (i.e., integer). The preceding statement reserves a port where the server waits for connections from clients. Each client asks to connect to the server on this port. Method **bind** raises the exception **socket**.**error** if the port is already in use, the hostname is incorrect or the port is reserved.

Software Engineering Observation 20.1

Port numbers can be between 0 and 65535. Many operating systems reserve port numbers below 1024 for system services (such as email and World Wide Web servers). Generally, these ports should not be specified as connection ports in user programs. In fact, some operating systems require special access privileges to use port numbers below 1024.

Common Programming Error 20.1

Specifying a port that is already in use or specifying an invalid port number when creating a ***socket*** *results in an error.*

The **socket** instance is now ready to receive a connection. In order to do so, the **socket** must prepare for a connection (step 3). This is done with a call to **socket** method **listen** of the form

```
socket.listen( backlog )
```

where *backlog* specifies the maximum number of clients that can request connections to the server. This value should be at least 1. As connections are received, they are queued. If the queue is full, client connections are refused.

The server **socket** then waits for a client to connect (step 4) with a call to **socket** method **accept**

```
connection, address = socket.accept()
```

The **socket** waits indefinitely (or *blocks*) when it calls method **accept**. When a client requests a connection, the method accepts the connection and returns to the server. Method **accept** returns a two-element tuple of the form (*connection*, *address*). The first element of the returned tuple (*connection*) is a new **socket** object that the server uses to commu-

nicate with the client. The second element (*address*) corresponds to the client's Internet address.

Step 5 is the processing phase in which the server and the client communicate. The server sends information to the client by invoking **socket** method **send** and passing the information in the form of a string. Method **send** returns the number of bytes sent. The server receives information from the client with **socket** method **recv**. When calling **recv**, the server must specify an integer that corresponds to the maximum amount of data that can be received at once. Method **recv** returns a string representing the received data. If the amount of data sent is greater than **recv** allows, the data is truncated and **recv** returns the maximum amount of data allowed. The excess data is buffered on the receiving end. On a subsequent call to **recv**, the excess data is removed from the buffer (along with any additional data the client may have sent since the previous call to **recv**).

Common Programming Error 20.2

A socket's **send** *method accepts only a string argument. Trying to pass a value with a different type (e.g., an integer) results in an error.*

In step 6, when the transmission is complete, the server closes the connection by invoking the **close** method on the **socket**.

Software Engineering Observation 20.2

With Python's multithreading capabilities, we can easily create multithreaded servers that can manage many simultaneous connections with many clients; this multithreaded-server architecture is precisely what is used in popular UNIX, Windows NT and OS/2 network servers.

Software Engineering Observation 20.3

A multithreaded server can be implemented to take the **socket** *returned by each call to* **accept** *and create a new thread that would manage network I/O across that* **socket***, or a multithreaded server can be implemented to maintain a pool of threads ready to manage network I/O across the new* **socket***s as they are created.*

Performance Tip 20.2

In high-performance systems in which memory is abundant, a multithreaded server can be implemented to create a pool of threads that can be assigned quickly to handle network I/O across each new **socket** *as it is created. Thus, when a connection is received, the server need not incur the overhead of thread creation.*

20.4 Establishing a Simple Client (Using Stream Sockets)

In this section, we discuss how to create a client that communicates with a server through a socket. Establishing a simple client in Python requires four steps. Step 1 creates a **socket** to connect to the server.

```
socket = socket.socket( family, type )
```

Step 2 connects to the server using **socket** method **connect**. Method **connect** takes as input the address of the socket to connect to. For **AF_INET** client sockets, the call to **connect** has the form

```
socket.connect( ( host, port ) )
```

where *host* is a string representing the server's hostname or IP address, and *port* is the integer port number that corresponds to the server process. If the connection attempt is successful, the client can now communicate with the server over the **socket**. A connection attempt that fails raises the **socket.error** exception.

Common Programming Error 20.3

A ***socket.error*** *exception is raised when a server address indicated by a client cannot be resolved or when an error occurs while attempting to connect to a server.*

Step 3 is the processing phase in which the client and the server communicate via methods **send** and **recv**. In step 4 when the transmission is complete, the client closes the connection by invoking the **close** method on the **socket**.

20.5 Client/Server Interaction with Stream Socket Connections

We now present an example (Fig. 20.2 and Fig. 20.3) that uses *stream sockets* to demonstrate a simple *client/server chat application*. The server waits for a client connection attempt. When a client application connects to the server, the server application sends a string to the client indicating that the connection was successful, and the client displays the message. Both the client and the server applications allow the user to type a message and send it to the other application. When the client or the server sends the string **"TERMINATE"**, the connection between the client and the server terminates. The client process terminates, and the server waits for the next client to connect. Figure 20.2 contains the definition of the server. The definition of the client is given in Fig. 20.3. Sample output showing the execution between the client and the server is shown as part of Fig. 20.3.

```
# Fig. 20.2: fig20_02.py
# Set up a server that will receive a connection
# from a client, send a string to the client,
# and close the connection

import socket

HOST = "127.0.0.1"
PORT = 5000
counter = 0

# step 1: create a socket
mySocket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

# step 2: bind the socket
mySocket.bind( ( HOST, PORT ) )

while 1:

   # step 3: prepare for a connection
```

Fig. 20.2 The server portion of a stream socket connection between a client and a server (part 1 of 2).

```
21    print "Waiting for connection"
22    mySocket.listen( 1 )
23
24    # step 4: wait for and accept a connection
25    connection, address = mySocket.accept()
26    counter += 1
27    print "Connection", counter, "received from:", address[ 0 ]
28
29    # step 5: process connection
30    connection.send( "SERVER>>> Connection successful" )
31    clientMessage = connection.recv( 1024 )
32
33    while clientMessage != "CLIENT>>> TERMINATE":
34
35       if not clientMessage:
36          break
37
38       print clientMessage
39       serverMessage = raw_input( "SERVER>>> " )
40       connection.send( "SERVER>>> " + serverMessage )
41       clientMessage = connection.recv( 1024 )
42
43    # step 6: close connection
44    print "Connection terminated"
45    connection.close()
```

Fig. 20.2 The server portion of a stream socket connection between a client and a server (part 2 of 2).

Lines 13–40 set up the server to receive a connection and to process the connection when it is received. Line 13 creates **socket** object **mySocket** to wait for connections. Integer **counter** (line 10) keeps track of the total number of connections processed.

Line 16 binds **mySocket** to port **5000**. Note that **HOST** is the string **"127.0.0.1"**. This causes the socket to use **localhost**, the hostname that corresponds to the machine on which the program is running. [*Note:* We chose to demonstrate the client/server relationship by connecting between programs executing on the same computer (localhost). Normally, this first argument would be a string containing the Internet address of another computer.] Lines 18–31 contain a **while** loop in which the server receives and processes each client connection. Line 22 listens for a connection from a client at port **5000**. The argument to **listen** is the number of connections that can wait in a queue to connect to the server (1 in this example). If the queue is full when a client requests a connection, the connection is refused.

Method **listen** sets up a listener to wait for a client connection. Once a connection is received, **socket** method **accept** (line 25) creates a **socket** object that manages the connection. Recall that **accept** returns a two-element tuple. The first element is a new **socket** instance that we call **connection**. The second element is the Internet address of the client computer that connected to this server (in the form (*host*, *port*) for **AF_INET** sockets). Once a new **socket** for the current connection exists, line 26 prints a message displaying the connection number and the client address.

Line 29 calls **socket** method **send** to send the string **"SERVER>>> Connection successful"** to the client. Line 30 calls **socket** method **recv** to receive a string from

the client of maximum size 1024 bytes. The **while** loop in lines 32–40 loops until the server receives the message **"CLIENT>>> TERMINATE"**. Lines 34–35 check whether the connection has been closed by the client. When a connection has been closed, **recv** returns an empty string. If this is the case, the **break** statement exits the loop. Otherwise, line 37 prints the message received from the client.

Function **raw_input** (line 38) reads a string from the user. The server sends this string to the client (line 39) and receives a message from the client (line 40). When the transmission is complete, line 44 closes the **socket**. The server awaits the next connection attempt from a client.

In our example, the server receives a connection, processes the connection, closes the connection and waits for the next connection. A more likely scenario would be a server that receives a connection, sets up that connection to be processed as a separate thread of execution and then waits for new connections. The separate threads that process existing connections can continue to execute while the server concentrates on new connection requests. We leave it as an exercise to implement this multithreaded approach to the server application.

The client is displayed in Fig. 20.3. Sample output from a client/server connection follows the code.

```
# Fig. 20.3: fig20_03.py
# Set up a client that will read information sent
# from a server and display that information

import socket

HOST = "127.0.0.1"
PORT = 5000

# step 1: create a socket
print "Attempting connection"
mySocket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

# step 2: connect to server
mySocket.connect( ( HOST, PORT ) )
print "Connected to Server"

# step 3: process connection
serverMessage = mySocket.recv( 1024 )

while serverMessage != "SERVER>>> TERMINATE":

   if not serverMessage:
      break

   print serverMessage
   clientMessage = raw_input( "CLIENT>>> " )
   mySocket.send( "CLIENT>>> " + clientMessage )
   serverMessage = mySocket.recv( 1024 )

```

Fig. 20.3 Demonstrating the client portion of a stream socket connection between a client and a server (part 1 of 2).

```
31   # step 4: close connection
32   print "Connection terminated"
33   mySocket.close()
```

```
Waiting for connection
Connection 1 received from: 127.0.0.1
```

```
Attempting connection
Connected to Server
SERVER>>> Connection successful
CLIENT>>> Hi to person at server
```

```
Waiting for connection
Connection 1 received from: 127.0.0.1
CLIENT>>> Hi to person at server
SERVER>>> Hi back to you--client!
```

```
Attempting connection
Connected to Server
SERVER>>> Connection successful
CLIENT>>> Hi to person at server
SERVER>>> Hi back to you--client!
CLIENT>>> TERMINATE
```

```
Waiting for connection
Connection 1 received from: 127.0.0.1
CLIENT>>> Hi to person at server
SERVER>>> Hi back to you--client!
Connection terminated
Waiting for connection
```

Fig. 20.3 Demonstrating the client portion of a stream socket connection between a client and a server (part 2 of 2).

Lines 12–29 perform the work necessary to connect to the server, to receive data from the server and to send data to the server. Line 12 creates a **socket** object—**mySocket**—to establish a connection. Line 15 attempts to connect to the server by calling **socket** method **connect** with one argument, a two-element tuple. Variable **PORT** is the same as in Fig. 20.2 (**5000**). This ensures that the client **socket** attempts to connect to the server on the port to which the server is bound.

If the connection is successful, line 16 prints a message to the screen. The **socket** method **recv** (line 19) receives a message from the server (i.e., **"SERVER>>> Connection successful"**). The **while** loop (lines 21–29) executes until the client receives the message **"SERVER>>> TERMINATE"**. As in the server program, line 23 checks each

received message to see if the server has closed the connection. If so, the **break** statement exits the **while** loop (line 24).

Each iteration of the loop prints the message received from the server and calls function **raw_input** to read a string from the user. Line 28 sends this string to the server by invoking **socket** method **send**. The client then receives the next message from the server (line 29). When the transmission is complete, line 33 closes the **socket** instance **mySocket**.

20.6 Connectionless Client/Server Interaction with Datagrams

We have been discussing *connection-oriented*, *streams-based transmission*. Now we consider c*onnectionless transmission with datagrams*.

Connection-oriented transmission is like the telephone system in which you dial and are given a *connection* to the telephone you wish to communicate with; the connection is maintained for the duration of your phone call, even when you are not talking.

Connectionless transmission with *datagrams* is more like the way mail is carried via the postal service. If a large message will not fit in one envelope, you break it into separate message pieces that you place in separate, sequentially numbered envelopes. Each of the letters is then mailed at once. The letters may arrive in order, out of order or not at all (although the last case is rare, it does happen). The person at the receiving end reassembles the message pieces into sequential order before attempting to make sense of the message. If your message is small enough to fit in one envelope, you do not have to worry about the “out-of-sequence” problem, but it is still possible that your message may not arrive. One difference between datagrams and postal mail is that duplicates of datagrams may arrive on the receiving computer.

The programs of Fig. 20.4 and Fig. 20.5 use datagrams to send packets of information between a client application and a server application. In the client application, the user types a message and presses *Enter*. The message is placed in a datagram packet that is sent to the server. The server receives the packet and displays the information in the packet, then *echoes* (copies) the packet back to the client. When the client receives the packet, the client displays the information in the packet. In this example, the client and server are implemented similarly.

```
# Fig. 20.4: fig20_04.py
# Set up a server that will receive packets from a
# client and send packets to a client.

import socket

HOST = "127.0.0.1"
PORT = 5000

# step 1: create a socket
mySocket = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )

# step 2: bind the socket
```

Fig. 20.4 The server side of a connectionless client/server computing with datagrams (part 1 of 2).

```
14 mySocket.bind( ( HOST, PORT ) )
15
16 while 1:
17
18    # step 3: receive packet
19    packet, address = mySocket.recvfrom( 1024 )
20
21    print "Packet received:"
22    print "From host:", address[ 0 ]
23    print "Host port:", address[ 1 ]
24    print "Length:", len( packet )
25    print "Containing:"
26    print "\t" + packet
27
28    # step 4: echo packet back to client
29    print "\nEcho data to client...",
30    mySocket.sendto( packet, address )
31    print "Packet sent\n"
32
33 mySocket.close()
```

```
Packet received:
From host: 127.0.0.1
Host port: 1645
Length: 20
Containing:
        first message packet

Echo data to client... Packet sent
```

Fig. 20.4 The server side of a connectionless client/server computing with datagrams (part 2 of 2).

The server (Fig. 20.4) defines one **socket** instance that sends and receives datagram (**SOCK_DGRAM**) packets. Note that the specified **socket** type is **SOCK_DGRAM**. This ensures that **mySocket** will be a datagram socket. Line 14 binds the socket to a port (**5000**) where packets can be received from clients. Clients sending packets to this server specify port **5000** in the packets they send.

The **while** loop in lines 16–31 receives packets from the client. First, line 19 waits for a packet to arrive. The ***recvfrom*** method blocks until a packet arrives. Once a packet arrives, **recvfrom** returns a string representing the data received and the address of the socket sending the data. The server then prints a message to the screen that contains the address of the client and the data sent.

Line 30 calls **socket** method **sendto** to echo the data back to the client. The method's first argument specifies the data to be sent. The second argument is a tuple that specifies the client computer's Internet address to which the packet will be sent and the port where the client is waiting to receive packets.

The client (Fig. 20.5) works similarly to the server, except that the client sends packets only when it is told to do so by the user typing a message and pressing the *Enter* key. The

while loop in lines 13–29 sends packets to the server using **sendto** (line 18) and waits for packets using **recvfrom** at line 22, which blocks until a packet arrives.

```
1  # Fig. 20.5: fig20_05.py
2  # Set up a client that will send packets to a
3  # server and receive packets from a server.
4
5  import socket
6
7  HOST = "127.0.0.1"
8  PORT = 5000
9
10 # step 1: create a socket
11 mySocket = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
12
13 while 1:
14
15    # step 2: send a packet
16    packet = raw_input( "Packet>>>" )
17    print "\nSending packet containing:", packet
18    mySocket.sendto( packet, ( HOST, PORT ) )
19    print "Packet sent\n"
20
21    # step 3: receive packet back from server
22    packet, address = mySocket.recvfrom( 1024 )
23
24    print "Packet received:"
25    print "From host:", address[ 0 ]
26    print "Host port:", address[ 1 ]
27    print "Length:", len( packet )
28    print "Containing:"
29    print "\t" + packet + "\n"
30
31 mySocket.close()
```

```
Packet>>>first message packet

Sending packet containing: first message packet
Packet sent

Packet received:
From host: 127.0.0.1
Host port: 5000
Length: 20
Containing:
        first message packet

Packet>>>
```

Fig. 20.5 Demonstrating the client side of a connectionless client/server computing with datagrams .

20.7 Client/Server Tic-Tac-Toe Using a Multithreaded Server

In this section, we present our capstone networking example—the popular game Tic-Tac-Toe implemented using client/server techniques with stream sockets. The program consists of a **`TicTacToeServer`** class (Fig. 20.6) that allows two **`TicTacToeClient`**s (Fig. 20.7) to connect to the server and play the game (outputs shown in Fig. 20.7). For each client connection, the server creates an instance of class **`Player`** (Fig. 20.6) to process the client in a separate thread of execution. This enables the clients to play the game independently. The first client to connect is automatically assigned Xs (X makes the first move) and the second client to connect is assigned Os. The server maintains the information about the game board so it can determine if a requested move by one of the players is valid or invalid. Each **`TicTacToeClient`** maintains its own GUI version of the Tic-Tac-Toe board on which the state of the game is displayed. The clients can only place a mark in an empty square on the board.

```python
# Fig. 20.6: fig20_06.py
# Class TicTacToeServer maintains a game of Tic-Tac-Toe
# for two clients, each managed by a Player thread.

import socket
import threading

class Player( threading.Thread ):
   "Thread used to manage each Tic-Tac-Toe client individually"

   def __init__( self, connection, server, number ):
      "Initialize thread and setup variables"

      threading.Thread.__init__( self )

      if number == 0:
         self.mark = "X"
      else:
         self.mark = "O"

      self.connection = connection
      self.server = server
      self.number = number

   def otherPlayerMoved( self, location ):
      "Notify client of opponent's last move"

      self.connection.send( "Opponent moved." )
      self.connection.send( str( location ) )

   def run( self ):
      "Play the game"

      self.server.display( "Player %s connected." % self.mark )
      self.connection.send( self.mark )
      self.connection.send( "%s connected." % self.mark )
```

Fig. 20.6 Server side of client/server Tic-Tac-Toe program (part 1 of 4).

```

      # wait for another player to arrive
      if self.mark == "X":
         self.connection.send( "Waiting for another player..." )
         self.server.gameBeginEvent.wait()
         self.connection.send(
            "Other player connected. Your move." )
      else:
         self.server.gameBeginEvent.wait()   # wait for server
         self.connection.send( "Waiting for first move..." )

      # play game
      while not self.server.gameOver():
         location = self.connection.recv( 2 )

         if not location:
            break

         if self.server.validMove( int( location ), self.number ):
            self.server.display( "loc: " + location )
            self.connection.send( "Valid move." )
         else:
            self.connection.send( "Invalid move, try again." )

      self.connection.close()
      self.server.display( "Game over." )
      self.server.display( "Connection closed." )

class TicTacToeServer:
   "Server that maintains a game of Tic-Tac-Toe for two clients"

   def __init__( self ):
      "Initialize variables and setup server"

      HOST = ""
      PORT = 5000

      self.board = []
      self.currentPlayer = 0
      self.turnCondition = threading.Condition()
      self.gameBeginEvent = threading.Event()

      for i in range( 9 ):
         self.board.append( None )

      # setup server socket
      self.server = socket.socket( socket.AF_INET,
         socket.SOCK_STREAM )
      self.server.bind( ( HOST, PORT ) )
      self.display( "Server awaiting connections..." )

   def execute( self ):
      "Play the game--create and start both Player threads"

```

Fig. 20.6 Server side of client/server Tic-Tac-Toe program (part 2 of 4).

```
      self.players = []

      for i in range( 2 ):
         self.server.listen( 1 )
         connection, address = self.server.accept()
         self.players.append( Player( connection, self, i ) )
         self.players[ -1 ].start()

      # players are suspended until player 0 connects
      # resume players now
      self.gameBeginEvent.set()

   def display( self, message ):
      "Display a message on the server"

      print message

   def validMove( self, location, player ):
      "Determine if a move is valid--if so, make move"

      # only one move can be made at a time
      self.turnCondition.acquire()

      while player != self.currentPlayer:
         self.turnCondition.wait()

      if not self.isOccupied( location ):

         if self.currentPlayer == 0:
            self.board[ location ] = "X"
         else:
            self.board[ location ] = "O"

         self.currentPlayer = ( self.currentPlayer + 1 ) % 2
         self.players[ self.currentPlayer ].otherPlayerMoved(
            location )
         self.turnCondition.notify()
         self.turnCondition.release()
         return 1
      else:
         self.turnCondition.notify()
         self.turnCondition.release()
         return 0

   def isOccupied( self, location ):
      "Determine if a space is occupied"

      return self.board[ location ]   # an empty space is None

   def gameOver( self ):
      "Determine if the game is over"

      # place code here testing for a game winner
      # left as an exercise for the reader
```

Fig. 20.6 Server side of client/server Tic-Tac-Toe program (part 3 of 4).

```
145         return 0
146
147 def main():
148    TicTacToeServer().execute()
149
150 if __name__ == "__main__":
151    main()
```

```
Server awaiting connections...
Player X connected.
Player O connected.
loc: 0
loc: 4
loc: 3
loc: 1
loc: 7
loc: 5
loc: 2
loc: 8
loc: 6
```

Fig. 20.6 Server side of client/server Tic-Tac-Toe program (part 4 of 4).

We begin with a discussion of the server side of the Tic-Tac-Toe game (Fig. 20.6). Line 148 instantiates a **TicTacToeServer** object and invokes its **execute** method. The **TicTacToeServer** constructor (lines 68–86) creates data member **currentPlayer** and condition variable **turnCondition**. The server uses these members to restrict access to method **validMove**—ensuring that only the current player can make a move. Line 77 creates **gameBeginEvent**—a **threading.Event** object used to synchronize the start of the game. Lines 79–80 then initialize the Tic-Tac-Toe board—a list of length 9. Note that each location of the board is initialized to **None**, indicating that the space is not yet occupied by either player. Locations are maintained as numbers from 0 to 8 (0 through 2 for the first row, 3 through 5 for the second row and 6 through 8 for the third row). Lines 83–86 prepare the **socket** on which the server listens for player connections and then display a message that the server is now ready.

Method **execute** (lines 88–101) loops twice, waiting each time for a connection from a client. When the server receives a connection, the server creates a new **Player** instance (lines 8–63) to manage the connection as a separate thread. The **Player** constructor (lines 11–23) takes as arguments the **socket** instance representing the connection to the client, the **TicTacToeServer** instance and a number indicating what player it is—X or O. Line 14 initializes the thread.

After the server creates each **Player** (line 96), the server invokes that instance's **start** method (line 97). The **Player**'s **run** method (lines 31–63) controls the information that is sent to and received from the client. First, the method passes to the client the character that the client places on the board when a move is made, then the method tells the client that a connection has been made (lines 35–36). Lines 39–43 then cause player X to block until the game can begin (i.e., player O has joined). Lines 44–46 similarly cause player O to block until the server begins the game. When both players have joined the game, the server starts the game by calling **Event** method **set** (line 101).

At this point, each **Player**'s **run** method executes its **while** loop (lines 49–59). Each iteration of this **while** loop receives a string representing the location where the client wants to place a mark and invokes **TicTacToeServer** method **validMove** to check the move. Lines 57 and 59 send a message to the client indicating whether or not the move was valid. The game continues until **TicTacToeServer** method **gameOver** (lines 140–145) indicates that the game is over. Lines 61–63 then close the connection to the client and display a message on the server.

Method **validMove** (lines 108–133 in class **TicTacToeServer**) uses condition variable methods **acquire** and **release** to allow only one move to be attempted at a time. This prevents both players from modifying the state information of the game simultaneously. If the **Player** attempting to validate a move is not the current player (i.e., the one allowed to make a move), the **Player** is placed in a *wait* state until it is that player's turn to move. If the position for the move being validated is already occupied on the board, the method returns 0. Otherwise, the server places a mark for the player in its local representation of the board, updates variable **currentPlayer**, calls **Player** method **otherPlayerMoved** (lines 25–29) so the client can be notified, invokes the **notify** method so the waiting **Player** (if there is one) can validate a move and returns 1 to indicate that the move is valid (lines 124–129).

When a **TicTacToeClient** (Fig. 20.7) begins execution, it creates a **Pmw ScrolledText** that displays messages from the server and creates a representation of the board using nine **Tkinter Button**s. Class **TicTacToeClient** inherits from class **threading.Thread** so that a separate thread can be used to continually read messages that are sent from the server to the client. The script's **run** method (lines 54–82) opens a connection to the server. After the client establishes a connection to the server, the method reads the mark character (X or O) from the server (line 65), initializes attribute **myTurn** to 0 (line 68) and loops continually to read messages from the server (lines 71–77). The messages are passed to the script's **processMessage** method for processing. When the game is over (i.e., the server closes the connection), lines 79–82 close the connection and display a message to the user.

```
1   # Fig. 20.7: fig20_07.py
2   # Client for Tic-Tac-Toe program
3
4   import socket
5   import threading
6   from Tkinter import *
7   import Pmw
8
9   class TicTacToeClient( Frame, threading.Thread ):
10     "Client that plays a game of Tic-Tac-Toe"
11
12     def __init__( self ):
13        "Create GUI and play game"
14
15        threading.Thread.__init__( self )
16
17        # initialize GUI
```

Fig. 20.7 Client side of a client/server Tic-Tac-Toe program (part 1 of 5).

```
      Frame.__init__( self )
      Pmw.initialise()
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Tic-Tac-Toe Client" )
      self.master.geometry( "250x325" )

      self.id = Label( self, anchor = W )
      self.id.grid( columnspan = 3, sticky = W+E+N+S )

      self.board = []

      # create and add all buttons to the board
      for i in range( 9 ):
         newButton = Button( self, font = "Courier 20 bold",
            height = 1, width = 1, relief = GROOVE,
            name = str( i ) )
         newButton.bind( "<Button-1>", self.sendClickedSquare )
         self.board.append( newButton )

      current = 0

      # display all buttons in 3x3 grid
      for i in range( 1, 4 ):

         for j in range( 3 ):
            self.board[ current ].grid( row = i, column = j,
               sticky = W+E+N+S )
            current += 1

      # area for server messages
      self.display = Pmw.ScrolledText( self, text_height = 10,
         text_width = 35, vscrollmode = "static" )
      self.display.grid( row = 4, columnspan = 3 )

      self.start()   # run thread

   def run( self ):
      "Control thread that allows continuous update of the display"

      # setup connection to server
      HOST = "127.0.0.1"
      PORT = 5000
      self.connection = socket.socket( socket.AF_INET,
         socket.SOCK_STREAM )
      self.connection.connect( ( HOST, PORT ) )

      # first get player's mark ( X or O )
      self.myMark = self.connection.recv( 2 )
      self.id.config( text = 'You are player "%s"' % self.myMark )

      self.myTurn = 0

      # receive messages sent to client
      while 1:
```

Fig. 20.7 Client side of a client/server Tic-Tac-Toe program (part 2 of 5).

```
            message = self.connection.recv( 34 )

            if not message:
               break

            self.processMessage( message )

         self.connection.close()
         self.display.insert( END, "Game over.\n" )
         self.display.insert( END, "Connection closed.\n" )
         self.display.yview( END )

      def processMessage( self, message ):
         "Interpret server message and perform necessary actions"

         if message == "Valid move.":
            self.display.insert( END, "Valid move, please wait.\n" )
            self.display.yview( END )
            self.board[ self.currentSquare ].config(
               text = self.myMark, bg = "white" )
         elif message == "Invalid move, try again.":
            self.display.insert( END, message + "\n" )
            self.display.yview( END )
            self.myTurn = 1
         elif message == "Opponent moved.":
            location = int( self.connection.recv( 2 ) )

            if self.myMark == "X":
               self.board[ location ].config( text = "O",
                  bg = "gray" )
            else:
               self.board[ location ].config( text = "X",
                  bg = "gray" )

            self.display.insert( END, message + " Your turn.\n" )
            self.display.yview( END )
            self.myTurn = 1
         elif message == "Other player connected. Your move.":
            self.display.insert( END, message + "\n" )
            self.display.yview( END )
            self.myTurn = 1
         else:
            self.display.insert( END, message + "\n" )
            self.display.yview( END )

      def sendClickedSquare( self, event ):
         "Send attempted move to server"

         if self.myTurn:
            name = event.widget.winfo_name()
            self.currentSquare = int( name )
            self.connection.send( name )
            self.myTurn = 0

```

Fig. 20.7 Client side of a client/server Tic-Tac-Toe program (part 3 of 5).

```python
def main():
    TicTacToeClient().mainloop()

if __name__ == "__main__":
    main()
```

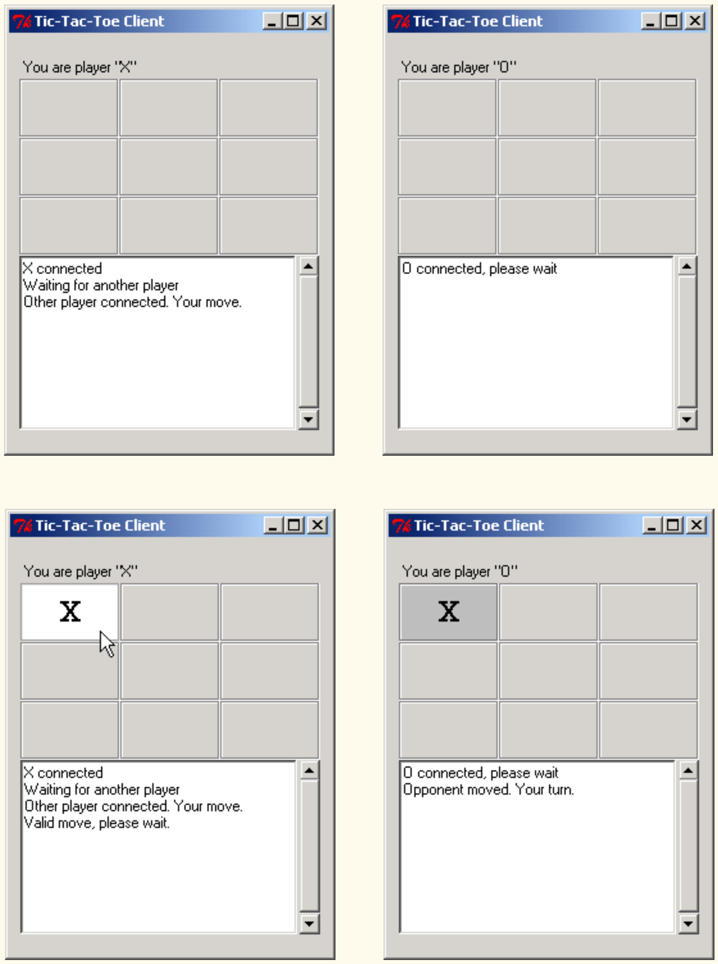


Fig. 20.7 Client side of a client/server Tic-Tac-Toe program (part 4 of 5).

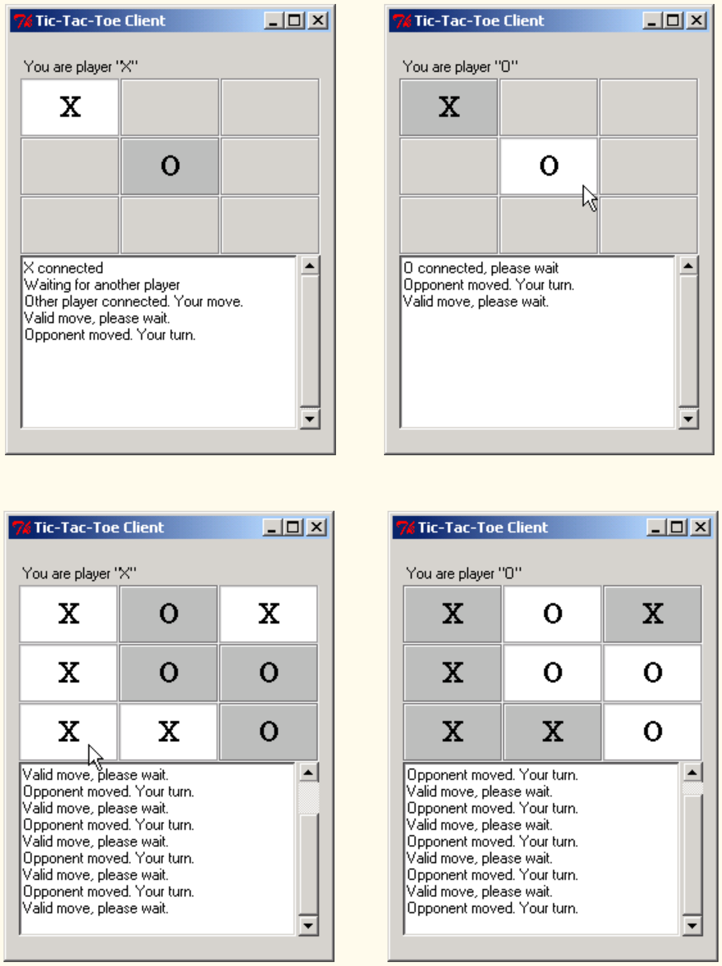


Fig. 20.7 Client side of a client/server Tic-Tac-Toe program (part 5 of 5).

Method **`processMessage`** (lines 84–115) interprets server messages. If the message received is the string **`"Valid move."`**, the client displays the message **`"Valid move, please wait."`**, sets its mark in the square that the user clicked (indicated by attribute **`currentSquare`**) and colors the square white. If the client receives the message **`"Invalid move, try again."`**, the client displays the message and sets attribute **`myTurn`** to 1 so the user can click a different square. If the client receives the message **`"Opponent moved."`**, the client receives an integer from the server indicating where the opponent moved. The client then places the opponent's mark in that square of the board, colors the square gray, displays a message and sets **`myTurn`** to 1. If the client receives the message **`"Other player connected. Your move."`**, the client displays the message and sets **`myTurn`** to 1. Note that this message is sent to player X only when player O

initially connects (lines 42–43). If the client receives any other message, the client simply displays the message.

When the player clicks a space on the board (a **`Tkinter Button`**), method **`send-ClickedSquare`** is invoked. Method **`sendClickedSquare`** (lines 117–124) first tests whether it is the player's turn. If so, line 121 obtains the name of the button pressed by invoking **`Widget`** method **`winfo_name`** and stores the value in variable **`name`**. Lines 122–124 then update attribute **`currentSquare`**, send the move to the server and set attribute **`myTurn`** to 0, so that the player cannot make another move until it has received feedback from the server.

SUMMARY

[*To be done for second round of review***]**

TERMINOLOGY

[*To be done for second round of review***]**

SELF-REVIEW EXERCISES

[*To be done for second round of review***]**

ANSWERS TO SELF-REVIEW EXERCISES

[*To be done for second round of review***]**

EXERCISES

[*To be done for second round of review***]**

Notes to Reviewers:

- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **`ben.wiedermann@deitel.net`**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copy edited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are mostly concerned with technical correctness and correct use of idiom. We will not make significant adjustments to our writing or coding style on a global scale. Please send us a short e-mail if you would like to make a suggestion.
- If you find something incorrect, please show us how to correct it.
- In the later round(s) of review, please read all the back matter, including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

A

B

C

D

E

F

H

I

L

M

N

P

Q

R

S

21

Security

Objectives

- To understand the basic concepts of security.
- To understand public-key/private-key cryptography.
- To learn about popular security protocols, such as SSL.
- To understand digital signatures, digital certificates, certificate authorities and public-key infrastructure.
- To understand Python programming security issues.
- To learn to write restricted Python code.
- To become aware of various threats to secure systems.

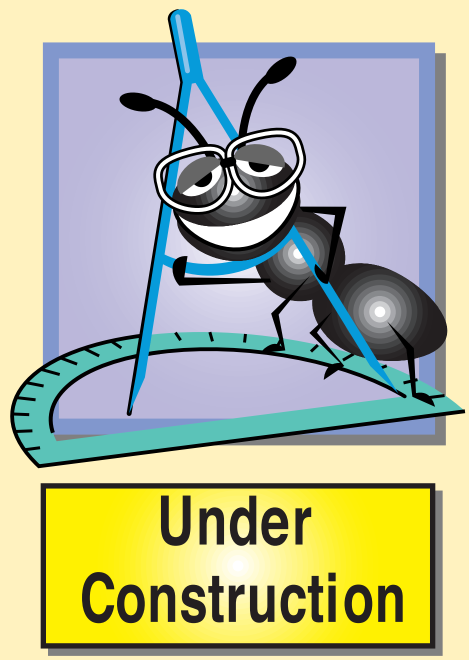
Three may keep a secret, if two of them are dead.
Benjamin Franklin

Attack—Repeat—Attack.
William Frederick Halsey, Jr.

Private information is practically the source of every large modern fortune.
Oscar Wilde

There must be security for all—or not one is safe.
The Day the Earth Stood Still, screenplay by Edmund H. North

No government can be long secure without formidable opposition.
Benjamin Disraeli



Outline

21.1 Introduction

The explosion of e-business is forcing companies and consumers to focus on Internet and network security. Consumers are buying products, trading stocks and banking online. They are submitting their credit-card numbers, social-security numbers and other confidential information to vendors through Web sites. Businesses are sending confidential information to clients and vendors using the Internet. At the same time, an increasing number of security attacks are taking place on e-businesses, and companies and customers are vulnerable to these attacks. Data theft and hacker attacks can corrupt files and even shut down businesses. Preventing or protecting against such attacks is crucial to the success of e-business. In this chapter, we explore Internet security, including securing electronic transactions and networks. We discuss how a Python programmer can secure programming code. We also examine the fundamentals of secure business and how to secure e-commerce transactions using current technologies.

e-Fact 21.1

According to a study by International Data Corporation (IDC), organizations spent $6.2 billion on security consulting in 1999, and IDC expects the market to reach $14.8 billion by 2003.[1]

Modern computer security addresses the problems and concerns of protecting electronic communications and maintaining network security. There are four fundamental requirements for a successful, secure transaction: *privacy*, *integrity*, *authentication* and *non-repudiation*. *The privacy issue is*: How do you ensure that the information you transmit over the Internet has not been captured or passed on to a third party without your knowledge? *The integrity issue is*: How do you ensure that the information you send or receive has not been compromised or altered? *The authentication issue is*: How do the sender and receiver of a message prove their identities to each other? *The nonrepudiation issue is*: How do you legally prove that a message was sent or received?

In addition to these requirements, network security addresses the issue of *availability*: How do we ensure that the network and the computer systems to which it connects will stay in continuous operation?

Python applications potentially can access files on the local computer on which the code is run. This chapter explains how a programmer can write secure, *restricted environment* Python code.

e-Fact 21.2

According to Forrester Research, it is predicted that organizations will spend 55% more on security in 2002 than they spent in 2000.[2]

We encourage you to visit the Web resources provided in Section 21.16 to learn more about the latest developments in e-business security. These resources include many informative and entertaining demos.

21.2 Ancient Ciphers to Modern Cryptosystems

The channels through which data passes are inherently unsecure; therefore, any private information passed through these channels must somehow be protected. To secure information, data can be encrypted. *Cryptography* transforms data by using a *cipher*, or

cryptosystem—a mathematical algorithm for encrypting messages (algorithm is a computer science term for "procedure"). A *key*—a string of digits that acts as a password—is input to the cipher. The cipher uses the key to make data incomprehensible to all but the sender and intended receivers. Unencrypted data is called *plaintext*; encrypted data is called *ciphertext*. The algorithm is responsible for encrypting data, while the key acts as a variable—using different keys results in different ciphertext. Only the intended receivers should have the corresponding key to decrypt the ciphertext into plaintext.

Cryptographic ciphers have been used throughout history, first recorded by the ancient Egyptians, to conceal and protect valuable information. In ancient cryptography, messages were encrypted by hand, usually with a method based on the alphabetic letters of the message. The two main types of ciphers were *substitution ciphers* and *transposition ciphers*. In a substitution cipher, every occurrence of a given letter is replaced by a different letter; for example, if every "a" is replaced by a "b," every "b" by a "c," etc., the word "security" would encrypt to "tfdvsjuz." The first prominent substitution cipher was credited to Julius Caesar, and is referred to today as the *Caesar Cipher*. Using the Caesar Cipher, every instance of a letter is encrypted by replacing by the letter in the alphabet three places to the right. For example, using the Caesar Cipher, the word "security" would encrypt to "vhfxulwb."

In a transposition cipher, the ordering of the letters is shifted; for example, if every other letter, starting with "s," in the word "security" creates the first word in the ciphertext and the remaining letters create the second word in the ciphertext, the word "security" would encrypt to "scrt euiy." Complicated ciphers combine substitution and transposition ciphers. For example, using the substitution cipher first, followed by the transposition cipher, the word "security" would encrypt to "tdsu fvjz." The problem with many historical ciphers is that their security relied on the sender and receiver to remember the encryption algorithm and keep it secret. Such algorithms are called *restricted algorithms*. Restricted algorithms are not feasible to implement among a large group of people. Imagine if the security of U.S. government communications relied on every U.S. government employee to keep a secret; the encryption algorithm could easily be compromised.

Modern cryptosystems are digital. Their algorithms are based on the individual *bits* or *blocks* (a group of bits) of a message, rather than letters of the alphabet. A computer stores data as a *binary string*, which is a sequence of ones and zeros. Each digit in the sequence is called a bit. Encryption and decryption keys are binary strings with a given *key length*. For example, 128-bit encryption systems have a key length of 128 bits. Longer keys have stronger encryption; it takes more time and computing power to crack the message.

Until January 2000, the U.S. government placed restrictions on the strength of cryptosystems that could be exported from the United States by limiting the key length of the encryption algorithms. Today, the regulations on exporting products that employ cryptography are less stringent. Any cryptography product may be exported as long as the end user is not a foreign government or from a country with embargo restrictions on it.[3]

21.3 Secret-key Cryptography

In the past, organizations wishing to maintain a secure computing environment used *symmetric cryptography*, also known as *secret-key cryptography*. Secret-key cryptography uses the same secret key to encrypt and decrypt a message (Fig. 21.1). In this case, the send-

er encrypts a message using the secret key, then sends the encrypted message to the intended recipient. A fundamental problem with secret-key cryptography is that before two people can communicate securely, they must find a secure way to exchange the secret key. One approach is to have the key delivered by a courier, such as a mail service or FedEx. While this approach may be feasible when two individuals communicate, it is not efficient for securing communication in a large network, nor can it be considered completely secure. The privacy and the integrity of the message would be compromised if the key is intercepted as it is passed between the sender and the receiver over unsecure channels. Also, since both parties in the transaction use the same key to encrypt and decrypt a message, one cannot authenticate which party created the message. Finally, to keep communications private with each receiver, a sender needs a different secret key for each receiver. As a result, organizations would have huge numbers of secret keys to maintain.

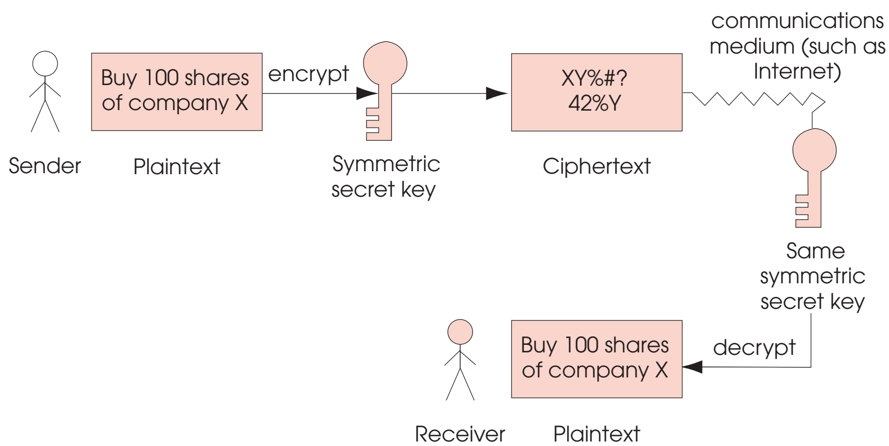


Fig. 21.1 Encrypting and decrypting a message using a secret key.

An alternative approach to the key-exchange problem is to have a central authority, called a *key distribution center* (*KDC*). The key distribution center shares a (different) secret key with every user in the network. In this system, the key distribution center generates a *session key* to be used for a transaction (Fig. 21.2). Next, the key distribution center distributes the session key to the sender and receiver, encrypted with the secret key they each share with the key distribution center. For example, say a merchant and a customer want to conduct a secure transaction. The merchant and the customer each have unique secret keys that they share with the key distribution center. The key distribution center generates a session key for the merchant and customer to use in the transaction. The key distribution center then sends the session key for the transaction to the merchant, encrypted using the secret key the merchant already shares with the center. The key distribution center sends the same session key for the transaction to the customer, encrypted using the secret key the customer already shares with the key distribution center. Once the merchant and the

customer have the session key for the transaction they can communicate with each other, encrypting their messages using the shared session key.

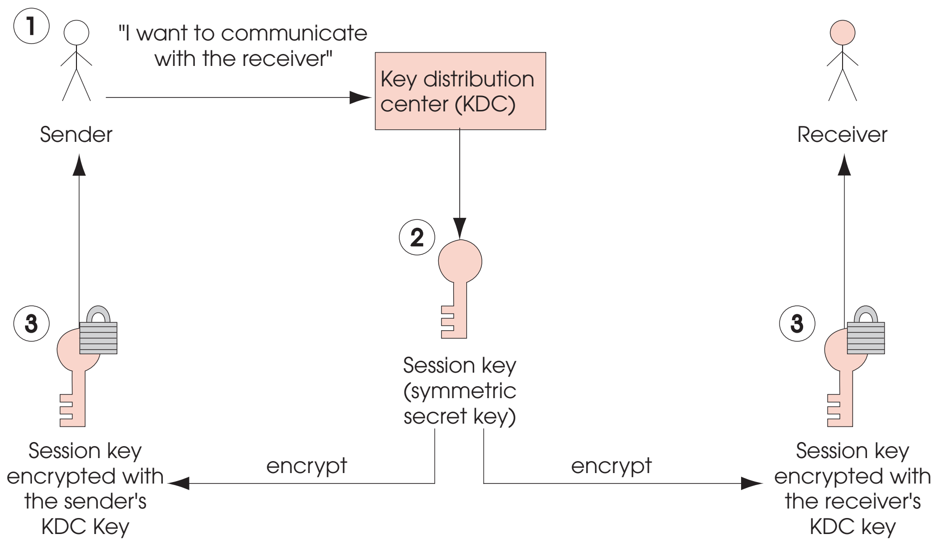


Fig. 21.2 Distributing a session key with a key distribution center.

Using a key distribution center reduces the number of courier deliveries (again, by means such as mail or FedEx) of secret keys to each user in the network. In addition, users can have a new secret key for each communication with other users in the network, which greatly increases the overall security of the network. However, if the security of the key distribution center is compromised, then the security of the entire network is compromised.

One of the most commonly used symmetric encryption algorithms is the *Data Encryption Standard* (*DES*). Horst Feistel of IBM created the *Lucifer* algorithm, which was chosen as the DES by the United States government and the National Security Agency (NSA) in the 1970s.[4] DES has a key length of 56 bits and encrypts data in 64-bit blocks. This type of encryption is known as a *block cipher*. A block cipher is an encryption method that creates groups of bits from an original message, then applies an encryption algorithm to the block as a whole, rather than as individual bits. This method reduces the amount of computer processing power and time required, while maintaining a fair level of security. For many years, DES was the encryption standard set by the U.S. government and the *American National Standards Institute* (*ANSI*). However, due to advances in technology and computing speed, DES is no longer considered secure. In the late 1990s, specialized *DES cracker machines* were built that recovered DES keys after just several hours.[5] As a result, the old standard of symmetric encryption has been replaced by *Triple DES*, or *3DES*, a variant of DES that is essentially three DES systems in a row, each with its own secret key. Though 3DES is more secure, the three passes through the DES algorithm result in slower performance. The United States government recently selected a new, more secure standard for symmetric encryption to replace DES. The new standard is called the *Advanced Encryption Standard* (*AES*). The *National Institute of Standards and Technology* (*NIST*), which sets the crypto-

graphic standards for the U.S. government, is evaluating *Rijndael* as the encryption method for AES. Rijndael is a block cipher developed by Dr. Joan Daemen and Dr. Vincent Rijmen of Belgium. Rijndael can be used with key sizes and block sizes of 128, 192 or 256 bits. Rijndael was chosen over four other finalists as the AES candidate because of its high security, performance, efficiency, flexibility and low memory requirement for computing systems.[6] For more information about AES, visit **csrc.nist.gov/encryption/aes**.

21.4 Public-key Cryptography

In 1976, Whitfield Diffie and Martin Hellman, researchers at Stanford University, developed *public-key cryptography* to solve the problem of exchanging keys securely. Public-key cryptography is asymmetric. It uses two inversely related keys: a *public key* and a *private key*. The private key is kept secret by its owner, while the public key is freely distributed. If the public key is used to encrypt a message, only the corresponding private key can decrypt it, and vice versa (Fig. 21.3). Each party in a transaction has both a public key and a private key. To transmit a message securely, the sender uses the receiver's public key to encrypt the message. The receiver then decrypts the message using his or her unique private key. Assuming that the private key has been kept secret, the message cannot be read by anyone other than the intended receiver. Thus the system ensures the privacy of the message. The defining property of a secure public-key algorithm is that it is "computationally infeasible" to deduce the private key from the public key. Although the two keys are mathematically related, deriving one from the other would take enormous amounts of computing power and time, enough to discourage attempts to deduce the private key. An outside party cannot participate in communication without the correct keys. The security of the entire process is based on the secrecy of the private keys. Therefore, if a third party obtains the private key used in decryption, the security of the whole system is compromised. If a system's integrity is compromised, the user can simply change the key, instead of changing the entire encryption or decryption algorithm.

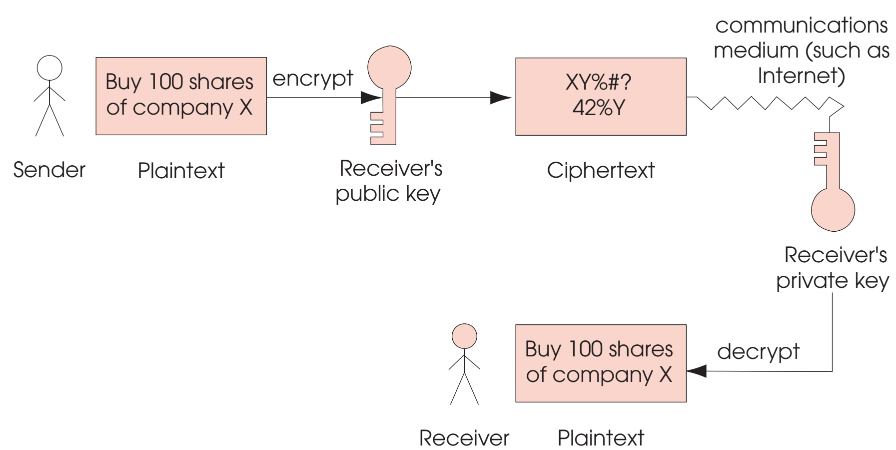


Fig. 21.3 Encrypting and decrypting a message using public-key cryptography.

Either the public key or the private key can be used to encrypt or decrypt a message. For example, if a customer uses a merchant's public key to encrypt a message, only the merchant can decrypt the message, using the merchant's private key. Thus, the merchant's identity can be authenticated, since only the merchant knows the private key. However, the merchant has no way of validating the customer's identity, since the encryption key the customer used is publicly available.

If the decryption key is the sender's public key and the encryption key is the sender's private key, the sender of the message can be authenticated. For example, suppose a customer sends a merchant a message encrypted using the customer's private key. The merchant decrypts the message using the customer's public key. Since the customer encrypted the message using his or her private key, the merchant can be confident of the customer's identity. This process authenticates the sender, but does not ensure confidentiality, as anyone could decrypt the message with the sender's public key. This systems works as long as the merchant can be sure that the public key with which the merchant decrypted the message belongs to the customer, and not a third party posing as the customer.

These two methods of public-key encryption can actually be used together to authenticate both participants in a communication (Fig. 21.4). Suppose a merchant wants to send a message securely to a customer so that only the customer can read it, and suppose also that the merchant wants to provide proof to the customer that the merchant (not an unknown third party) actually sent the message. First, the merchant encrypts the message using the customer's public key. This step guarantees that only the customer can read the message. Then the merchant encrypts the result using the merchant's private key, which proves the identity of the merchant. The customer decrypts the message in reverse order. First, the customer uses the merchant's public key. Since only the merchant could have encrypted the message with the inversely related private key, this step authenticates the merchant. Then the customer uses the customer's private key to decrypt the next level of encryption. This step ensures that the content of the message was kept private in the transmission, since only the customer has the key to decrypt the message. Although this system provides extremely secure transactions, the setup cost and time required prevent widespread use.

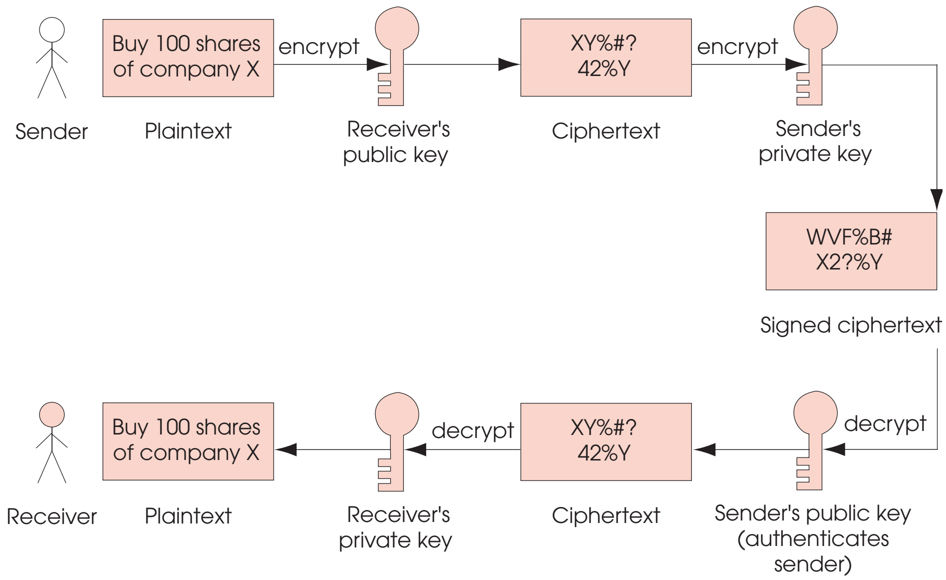


Fig. 21.4 Authentication with a public-key algorithm

The most commonly used public-key algorithm is *RSA*, an encryption system developed in 1977 by MIT professors Ron Rivest, Adi Shamir and Leonard Adleman.[7] Today, most Fortune 1000 companies and leading e-commerce businesses use their encryption and authentication technologies. With the emergence of the Internet and the World Wide Web, their security work has become even more significant and plays a crucial role in e-commerce transactions. Their encryption products are built into hundreds of millions of copies of the most popular Internet applications, including Web browsers, commerce servers and e-mail systems. Most secure e-commerce transactions and communications on the Internet use RSA products. For more information about RSA, cryptography and security, visit **www.rsasecurity.com**.

Pretty Good Privacy (PGP) is a public-key encryption system used for the encryption of e-mail messages and files. PGP was designed in 1991 by Phillip Zimmermann.[8] PGP can also be used to provide digital signatures (see Section 21.8, Digital Signatures) that confirm the author of an e-mail or public posting. PGP is based on a “web of trust;” each client in a network can vouch for another client’s identity to prove ownership of a public key. The “web of trust” is used to authenticate each client. If users know the identity of a public key holder, through personal contact or another secure method, they validate the key by signing it with their own key. The web grows as more users validate the keys of others. To learn more about PGP and to download a free copy of the software, go to the MIT Distribution Center for PGP at **web.mit.edu/network/pgp.html**.

21.5 Cryptanalysis

Even if keys are kept secret, it may be possible to compromise the security of a system. Trying to decrypt ciphertext without knowledge of the decryption key is known as *cryptanalysis*. Commercial encryption systems are constantly being researched by cryptologists to ensure that the systems are not vulnerable to a *cryptanalytic attack*. The most common form of cryptanalytic attacks are those in which the encryption algorithm is analyzed to find relations between bits of the encryption key and bits of the ciphertext. Often, these relations are only statistical in nature and incorporate an analyzer's outside knowledge about the plaintext. The goal of such an attack is to determine the key from the ciphertext.

Weak statistical trends between ciphertext and keys can be exploited to gain knowledge about the key if enough ciphertext is known. Proper key management and expiration dates on keys help prevent cryptanalytic attacks. When a key is used for long periods of time, more ciphertext is generated that can be beneficial to an attacker trying to derive a key. If a key is unknowingly recovered by an attacker, it can be used to decrypt every message for the life of that key. Using public-key cryptography to exchange secret keys securely allows a new secret key to encrypt every message.

21.6 Key Agreement Protocols

A drawback of public-key algorithms is that they are not efficient for sending large amounts of data. They require significant computer power, which slows down communication. Public-key algorithms should not be thought of as a replacement for secret-key algorithms. Instead, public-key algorithms allow two parties to agree on a key to be used for secret-key encryption over an unsecure medium. The process by which two parties can exchange keys over an unsecure medium is called a *key agreement protocol*. A *protocol* sets the rules for communication: Exactly what encryption algorithm(s) is (are) going to be used?

The most common key agreement protocol is a *digital envelope* (Fig. 21.5). With a digital envelope, the message is encrypted using a secret key (Step 1), and the secret key is encrypted using public-key encryption (Step 2). The sender attaches the encrypted secret key to the encrypted message and sends the receiver the entire package. The sender could also digitally sign the package before sending it to prove the sender's identity to the receiver (Section 23.8). To decrypt the package, the receiver first decrypts the secret key using the receiver's private key. Then, the receiver uses the secret key to decrypt the actual message. Since only the receiver can decrypt the encrypted secret key, the sender can be sure that only the intended receiver is reading the message.

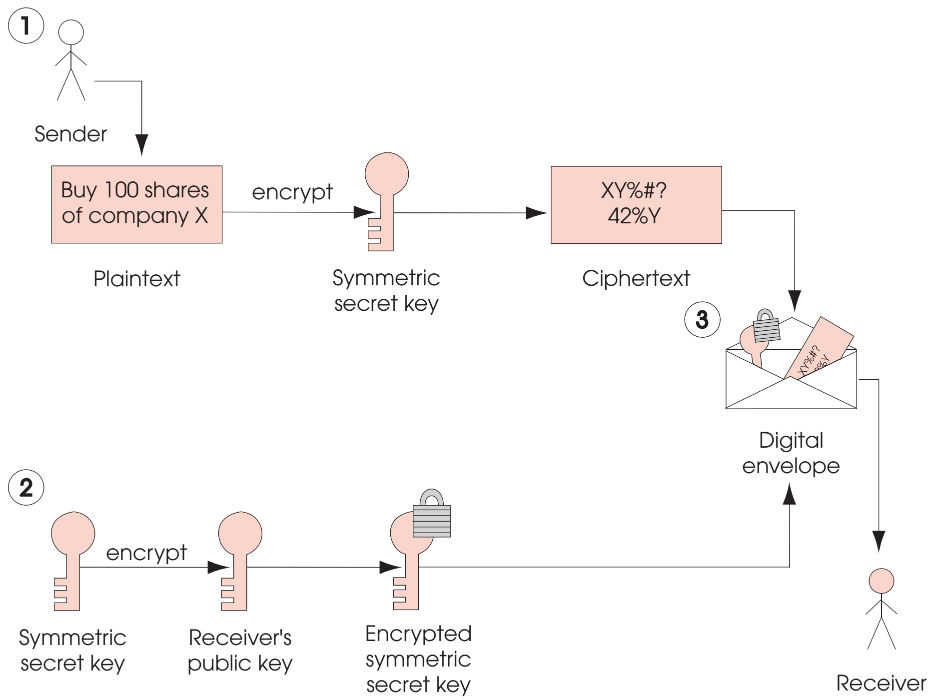


Fig. 21.5 Creating a digital envelope.

21.7 Key Management

Maintaining the secrecy of private keys is crucial to keeping cryptographic systems secure. Most compromises in security result from poor *key management* (e.g., the mishandling of private keys, resulting in key theft) rather than attacks that attempt to guess the keys.[9]

A main component of key management is *key generation*—the process by which keys are created. A malicious third party could try to decrypt a message by using every possible decryption key, a process known as *brute-force cracking*. Key-generation algorithms are sometimes unintentionally constructed to choose from only a small subset of possible keys. If the subset is too small, then the encrypted data is more susceptible to brute-force attacks. Therefore, it is important to have a key-generation program that can generate a large number of keys as randomly as possible. Keys are made more secure by choosing a key length so large that it is computationally infeasible to try all combinations.

21.8 Digital Signatures

Digital signatures, the electronic equivalent of written signatures, were developed to be used in public-key cryptography to solve the problems of authentication and integrity (see Microsoft Authenticode feature). A digital signature authenticates the sender’s identity, and, like a written signature, it is difficult to forge.

To create a digital signature, a sender first takes the original plaintext message and runs it through a *hash function*, which is a mathematical calculation that gives the message a *hash value*. A *one-way hashing function* generates a string of characters that is unique to the input file. The *Secure Hash Algorithm* (*SHA-1*) is the current standard for hashing functions. In using SHA-1, the phrase “Buy 100 shares of company X” would produce the hash value *D8 A9 B6 9F 72 65 0B D5 6D 0C 47 00 95 0D FD 31 96 0A FD B5*. MD5 is another popular hash function, which was developed by Ronald Rivest to verify data integrity through a 128-bit hash value of the input file.[10] [***<**userpages.umbc.edu/~mabzug1/cs/md5/md5.html**>***] The following interactive session demonstrates the ways to get the MD5 hash of the same phrase in Python.

```
Python 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license" for more information.
>>> import md5
>>> m1 = md5.new( "Buy 100 shares of company X" )
>>> print m1.hexdigest()
de1746f8b9f91decab749e5fa3955af7
>>> m2 = md5.new()
>>> m2.update( "Buy 100 shares " )
>>> print m2.hexdigest()
1eaa9fd4f62aa1a88d64d2d69b7d4f13
>>> m2.update( "of company X" )
>>> print m2.hexdigest()
de1746f8b9f91decab749e5fa3955af7
>>>
```

Examples of SHA-1 and MD5 are available at **home.istar.ca/~neutron/messagedigest**. At this site, users can input text or files into a program to generate the hash value. The hash value is also known as a *message digest*. The chance that two different messages will have the same message digest is statistically insignificant. *Collision* occurs when multiple messages have the same hash value. It is computationally infeasible to compute a message from its hash value or to find two messages with the same hash value.

Next, the sender uses the sender’s private key to encrypt the message digest. This step creates a digital signature and authenticates the sender, since only the owner of that private key could encrypt the message. A message that includes the digital signature, hash function and original message (encrypted using the receiver’s public key) is sent to the receiver. The receiver uses the sender’s public key to decipher the original digital signature and reveal the message digest. The receiver then uses his or her own private key to decipher the original message. Finally, the receiver applies the hash function to the original message. If the hash value of the original message matches the message digest included in the signature, there is *message integrity*; the message has not been altered in transmission.

There is a fundamental difference between digital signatures and handwritten signatures. A handwritten signature is independent of the document being signed. Thus, if someone can forge a handwritten signature, they can use that signature to forge multiple documents. A digital signature is created using the contents of the document. Therefore, your digital signature is different for each document you sign.

Digital signatures do not provide proof that a message has been sent. Consider the following situation: A contractor sends a company a digitally signed contract, which the contractor later would like to revoke. The contractor could do so by releasing the private key and then claiming that the digitally signed contract came from an intruder who stole the contractor's private key. *Timestamping*, which binds a time and date to a digital document, can help solve the problem of non-repudiation. For example, suppose the company and the contractor are negotiating a contract. The company requires the contractor to sign the contract digitally and then have the document digitally timestamped by a third party called a *timestamping agency*. The contractor sends the digitally-signed contract to the timestamping agency. The privacy of the message is maintained since the timestamping agency sees only the encrypted, digitally-signed message (as opposed to the original plaintext message). The timestamping agency affixes the time and date of receipt to the encrypted, signed message and digitally signs the whole package with the timestamping agency's private key. The timestamp cannot be altered by anyone except the timestamping agency, since no one else possesses the timestamping agency's private key. Unless the contractor reports the private key to have been compromised before the document was timestamped, the contractor cannot legally prove that the document was signed by an unauthorized third party. The sender could also require the receiver to sign the message digitally and timestamp it as proof of receipt. To learn more about timestamping, visit **`AuthentiDate.com`**.

The U.S. government's digital-authentication standard is called the *Digital Signature Algorithm* (*DSA*). The U.S. government recently passed digital-signature legislation that makes digital signatures as legally binding as handwritten signatures. This legislation is expected to increase e-business dramatically. For the latest news about U.S. government legislation in information security, visit **`www.itaa.org/infosec`**. For more information about the bills, visit the following government sites:

```
thomas.loc.gov/cgi-bin/bdquery/z?d106:hr.01714:
thomas.loc.gov/cgi-bin/bdquery/z?d106:s.00761:
```

21.9 Public-key Infrastructure, Certificates and Certificate Authorities

One problem with public-key cryptography is that anyone with a set of keys could potentially assume another party's identity. For example, say a customer wants to place an order with an online merchant. How does the customer know that the Web site indeed belongs to that merchant and not to a third party that posted a site and is *masquerading* as a merchant to steal credit-card information? *Public Key Infrastructure* (*PKI*) provides a solution to these problems. PKI integrates public-key cryptography with *digital certificates* and *certificate authorities* to authenticate parties in a transaction.

e-Fact 21.3

The Aberdeen Group predicts that approximately 98% of all Global 2000 companies will implement PKI solutions by 2003.[11]

A digital certificate is a digital document used to identify a user and issued by a certificate authority (CA). A digital certificate includes the name of the subject (the company or individual being certified), the subject's public key, a serial number, an expiration date, the signature of the trusted certificate authority and any other relevant information (Fig. 21.6).

A CA is a financial institution or other trusted third party, such as *VeriSign*. Once issued, the digital certificates are publicly available and are held by the certificate authority in *certificate repositories*.

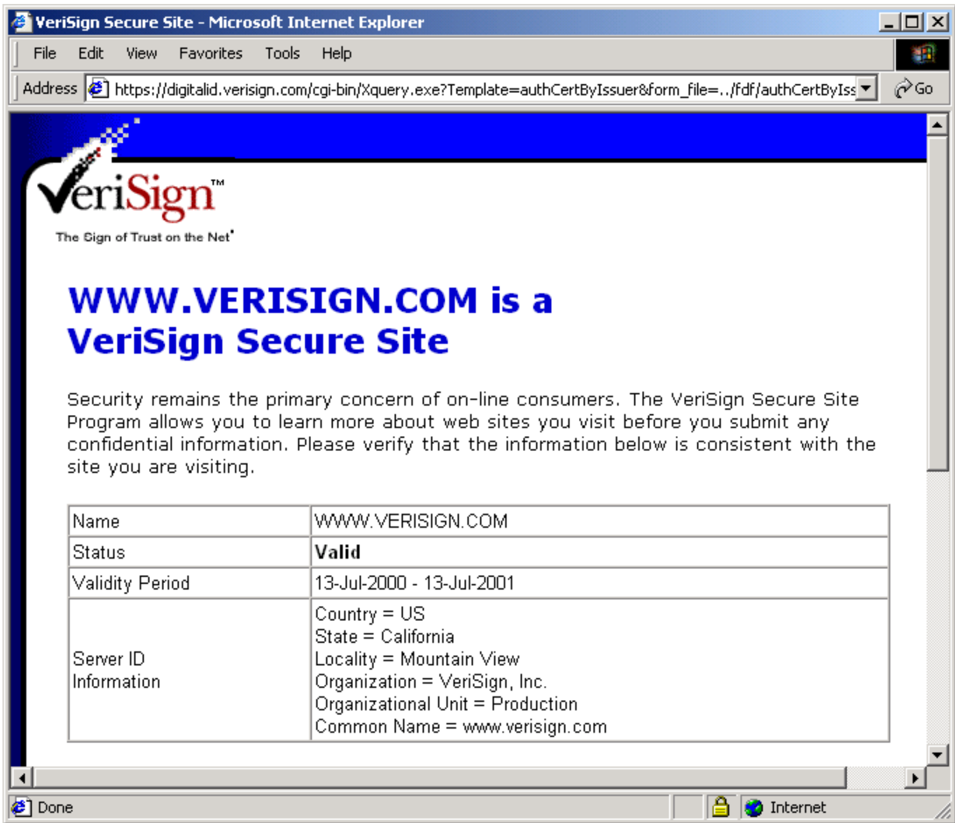


Fig. 21.6 Portion of a VeriSign digital certificate. (Courtesy of VeriSign, Inc.)

The CA signs the certificate by encrypting either the subject's public key or a hash value of the public key using the CA's own private key. The CA has to verify every subject's public key. Thus, users must trust the public key of a CA. Usually, each CA is part of a *certificate authority hierarchy*. This hierarchy is similar to a chain of trust in which each link relies on another link to provide authentication information. A certificate authority hierarchy is a chain of certificate authorities, starting with the *root certificate authority*, which is the Internet Policy Registration Authority (IPRA). The IPRA signs certificates using the *root key*. The root key signs certificates only for *policy creation authorities*, which are organizations that set policies for obtaining digital certificates. In turn, policy creation authorities sign digital certificates for CAs. CAs then sign digital certificates for individuals and organizations. The CA takes responsibility for authentication, so it must check information carefully before issuing a digital certificate. In one case, human error caused VeriSign to issue two digital certificates to an imposter posing as a Microsoft

employee.[12] Such an error is significant; the inappropriately issued certificates can cause users to download malicious code unknowingly onto their machines (see Authentication: Microsoft Authenticode feature).

VeriSign, Inc., is a leading certificate authority. For more information about VeriSign, visit **www.verisign.com**. For a listing of other digital-certificate vendors, please see Section 21.16.

e-Fact 21.4

It can take a year and cost from $5 million to $10 million for a financial firm to build a digital certificate infrastructure, according to Identrus, a consortium of global financial companies that is providing a framework for trusted business-to-business e-commerce.[13]

Periodically changing key pairs is necessary in maintaining a secure system, as a private key may be compromised without a user's knowledge. The longer a key pair is used, the more vulnerable the keys are to attack and cryptanalysis. As a result, digital certificates are created with an expiration date, to force users to switch key pairs. If a private key is compromised before its expiration date, the digital certificate can be canceled, and the user can get a new key pair and digital certificate. Canceled and revoked certificates are placed on a *certificate revocation list* (*CRL*). CRLs are stored with the certificate authority that issued the certificates. It is essential for users to report immediately if they suspect that their private keys have been compromised, as the issue of non-repudiation makes certificate owners responsible for anything appearing with their digital signatures. In states with laws on digital signatures, certificates legally bind certificate owners to any transactions involving their certificates.

CRLs are similar to old paper lists of revoked credit-card numbers that were used at the points of sale in stores.[14] This makes for a great inconvenience when checking the validity of a certificate. An alternative to CRLs is the *Online Certificate Status Protocol* (*OCSP*), which validates certificates in real-time. OCSP technology is currently under development. For an overview of OCSP, read "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP" located at **ftp.isi.edu/in-notes/rfc2560.txt**.

Many people still consider e-commerce unsecure. However, transactions using PKI and digital certificates can be more secure than exchanging private information over phone lines, through the mail or even than paying by credit card in person. After all, when you go to a restaurant and the waiter takes your credit card in back to process your bill, how do you know the waiter did not write down your credit-card information? In contrast, the key algorithms used in most secure online transactions are nearly impossible to compromise. By some estimates, the key algorithms used in public-key cryptography are so secure that even millions of today's computers working in parallel could not break the codes in a century. However, as computing power increases, key algorithms considered strong today could be broken in the future.

Digital-certificate capabilities are built into many e-mail packages. For example, in Microsoft Outlook, you can go to the **Tools** menu and select **Options**. Then click on the **Security** tab. At the bottom of the dialog box, you will see the option to obtain a digital ID. Selecting the option will take you to a Microsoft Web site with links to several worldwide certificate authorities. Once you have a digital certificate, you can sign your e-mail messages digitally.

To obtain a digital certificate for your personal e-mail messages, visit **www.verisign.com** or **www.thawte.com**. VeriSign offers a free 60-day trial, or you can purchase the service for a yearly fee. Thawte offers free digital certificates for personal e-mail. Web server certificates may also be purchased through VeriSign and Thawte; however, they are more expensive than e-mail certificates.

Authentication: Microsoft Authenticode

How do you know that the software you ordered online is safe and has not been altered? How can you be sure that you are not downloading a computer virus that could wipe out your computer? Do you trust the source of the software? With the emergence of e-commerce, software companies are offering their products online, so that customers can download software directly onto their computers. Security technology is used to ensure that the downloaded software is trustworthy and has not been altered. *Microsoft Authenticode*, combined with VeriSign digital certificates (or *digital IDs*), authenticates the publisher of the software and detects whether the software has been altered. Authenticode is a security feature built into Microsoft Internet Explorer.

To use Microsoft Authenticode technology, each software publisher must obtain a digital certificate specifically designed for the purpose of publishing software; such certificates may be obtained through certificate authorities, such as VeriSign (Section 6.9). To obtain a certificate, a software publisher must provide its public key and identification information and sign an agreement that it will not distribute harmful software. This requirement gives customers legal recourse if any downloaded software from certified publishers causes harm.

Microsoft Authenticode uses digital-signature technology to sign software (Section 6.8). The signed software and the publisher's digital certificate provide proof that the software is safe and has not been altered.

When a customer attempts to download a file, a dialog box appears on the screen displaying the digital certificate and the name of the certificate authority. Links to the publisher and the certificate authority are provided so that customers can learn more about each party before they agree to download the software. If Microsoft Authenticode determines that the software has been compromised, the transaction is terminated.

To learn more about Microsoft Authenticode, visit the following sites:

msdn.microsoft.com/workshop/security/authcode/signfaq.asp
msdn.microsoft.com/workshop/security/authcode/authwp.asp

21.9.1 Smart Cards

One of the fastest growing applications of PKI is the *smart card*. A smart card generally looks like a credit card and can serve many different functions, from authentication to data storage. The most popular smart cards are *memory cards* and *microprocessor cards*. Memory cards are similar to floppy disks. Microprocessor cards are similar to small computers, with operating systems, security and storage. Smart cards also have different *interfaces* with which they interact with reading devices. One type of interface is a *contact interface*, in which smart cards are inserted into a reading device and physical contact between the

device and the card is necessary. The alternative to this method is a *contactless interface,* in which data is transferred to a reader via an embedded wireless device in the card, without the card and the device having to make physical contact.[15]

Smart cards store private keys, digital certificates and other information necessary for implementing PKI. They may also store credit card numbers, personal contact information, etc. Each smart card is used in combination with a *personal identification number* (*PIN*). This application provides two levels of security by requiring the user to both possess a smart card and know the corresponding PIN to access the information stored on the card. As an added measure of security, some microprocessor cards will delete or corrupt stored data if malicious attempts at tampering with the card occur. Smart card PKI is stored portable, allowing users to access information from multiple devices using the same smart card.

e-Fact 21.5

According to Dataquest, use of smart cards is growing 30% per year, and it is expected that 3.4 billion smarts cards will be in used worldwide in 2001. [16]

21.10 Security Protocols

Everyone using the Web for e-business and e-commerce needs to be concerned about the security of their personal information. In this section, we discuss network security protocols, such as *Internet Protocol Security (IPSec),* and transport layer security protocols such as *Secure Sockets Layer* (*SSL*). Network security protocols protect communications between networks; transport layer security protocols are used to establish secure connections for data to pass through.

21.10.1 Secure Sockets Layer (SSL)

Currently, most e-businesses use SSL for secure online transactions, although SSL is not designed specifically for securing transactions. Rather, SSL secures World Wide Web connections. The Secure Sockets Layer (SSL) protocol, developed by Netscape Communications, is a non-proprietary protocol commonly used to secure communication between two computers on the Internet and the Web.[17] SSL is built into many Web browsers, including Netscape Communicator and Microsoft Internet Explorer, as well as numerous other software products. It operates between the Internet's TCP/IP communications protocol and the application software.[18]

In a standard correspondence over the Internet, a sender's message is passed to a *socket*, which receives and transmits information from a network. The socket then interprets the message through *Transmission Control Protocol/Internet Protocol* (*TCP/IP*). TCP/IP is the standard set of protocols used for connecting computers and networks to a network of networks, known as the Internet. Most Internet transmissions are sent as sets of individual message pieces, called *packets*. At the sending side, the packets of one message are numbered sequentially, and error-control information is attached to each packet. IP is primarily responsible for routing packets to avoid traffic jams, so each packet might travel a different route over the Internet. The destination of a packet is determined by the *IP address*—an assigned number used to identify a computer on a network, similar to the address of a house in a neighborhood. At the receiving end, the TCP makes sure that all of

the packets have arrived, puts them in sequential order and determines if the packets have arrived without alteration. If the packets have been accidentally altered or any data has been lost, TCP requests retransmission. However, TCP is not sophisticated enough to determine if packets have been maliciously altered during transmission, as malicious packets can be disguised as valid ones. When all of the data successfully reaches TCP/IP, the message is passed to the socket at the receiver end. The socket translates the message back into a form that can be read by the receiver's application.[19] In a transaction using SSL, the sockets are secured using public-key cryptography.

SSL implements public-key technology using the RSA algorithm and digital certificates to authenticate the server in a transaction and to protect private information as it passes from one party to another over the Internet. SSL transactions do not require client authentication; many servers consider a valid credit-card number to be sufficient for authentication in secure purchases. To begin, a client sends a message to a server. The server responds and sends its digital certificate to the client for authentication. Using public-key cryptography to communicate securely, the client and server negotiate *session keys* to continue the transaction. Session keys are secret keys that are used for the duration of that transaction. Once the keys are established, the communication proceeds between the client and the server by using the session keys and digital certificates. Encrypted data is passed through TCP/IP, just as regular packets travel over the Internet. However, before sending a message with TCP/IP, the SSL protocol breaks the information into blocks, compresses it and encrypts it. Conversely, after the data reaches the receiver through TCP/IP, the SSL protocol decrypts the packets, then decompresses and assembles the data. These extra processes provide an extra layer of security between TCP/IP and applications. SSL is primarily used to secure *point-to-point connections*—transmissions of data from one computer to another.[20] SSL allows for the authentication of the server, the client, both or neither; in most Internet SSL sessions, only the server is authenticated. The Transport Layer Security (TLS) protocol, designed by the Internet Engineering Task Force, is similar to SSL. For more information on TLS, visit: **www.ietf.org/rfc/rfc2246.txt**.

Although SSL protects information as it is passed over the Internet, it does not protect private information, such as credit-card numbers, once the information is stored on the merchant's server. When a merchant receives credit-card information with an order, the information is often decrypted and stored on the merchant's server until the order is placed. If the server is not secure and the data is not encrypted, an unauthorized party can access the information. Hardware devices, such as *peripheral component interconnect (PCI) cards* designed for use in SSL transactions, can be installed on Web servers to process SSL transactions, thus reducing processing time and leaving the server free to perform other tasks.[21] Visit **www.sonicwall.com/products/trans.asp** for more information on these devices. For more information about the SSL protocol, check out the Netscape SSL tutorial at **developer.netscape.com/tech/security/ssl/protocol.html** and the Netscape Security Center site at **www.netscape.com/security/index.html**.

21.10.2 IPSec and Virtual Private Networks (VPN)

Networks allow organizations to link multiple computers together. *Local area networks* (*LANs*) connect computers that are physically close, generally in the same building. *Wide area networks* (*WANs*) are used to connect computers in multiple locations using private telephone lines or radio waves. Organizations are now taking advantage of the existing in-

frastructure of the Internet—the publicly available wires—to create *Virtual Private Networks* (*VPNs*), linking multiple networks, wireless users and other remote users. VPNs use the Internet infrastructure that is already in place, therefore they are more economical than private networks such as WANs.[22] The encryption allows for VPNs to provide the same services as private networks over a public network.

A VPN is created by establishing a secure *tunnel* through which data passes between multiple networks over the Internet. *IPSec (Internet Protocol Security)* is one of the technologies used to secure the tunnel through which the data passes, ensuring the privacy and integrity of the data, as well authenticating the users.[23] IPSec, developed by the *Internet Engineering Task Force* (*IETF*), uses public-key and symmetric key cryptography to ensure authentication of the users, data integrity and confidentiality. The technology takes advantage of the standard that is already in place, in which information travels between two networks over the Internet via the *Internet Protocol* (*IP*). Information sent using IP, however, can easily be intercepted. Unauthorized users can access the network by using a number of well-known techniques, such as *IP spoofing*—a method in which an attacker simulates the IP of an authorized user or host to get access to resources that would otherwise be off-limits. The SSL protocol enables secure, point-to-point connections between two applications; IPSec enables the secure connection of an entire network. The Diffie-Hellman and RSA algorithms are commonly used in the IPSec protocol for key exchange, and DES or 3DES are used for secret-key encryption (depending on system and encryption needs). An IP packet is encrypted, then sent inside a regular IP packet that creates the tunnel. The receiver discards the outer IP packet, then decrypts the inner IP packet.[24] VPN security relies on three concepts—authentication of the user, encryption of the data sent over the network and controlled access to corporate information.[25] To address these three security concepts, IPSec is composed of three pieces. The *Authentication Header* (*AH*) attaches additional information to each packet, which verifies the identity of the sender and proves that data was not modified in transit. The *Encapsulating Security Payload* (*ESP*) encrypts the data using symmetric key ciphers to protect the data from eavesdroppers while the IP packet is being sent from one computer to another. The *Internet Key Exchange* (*IKE*) is the key-exchange protocol used in IPSec to determine security restrictions and to authenticate the encryption keys.

VPNs are becoming increasingly popular in businesses. However, VPN security is difficult to manage. To establish a VPN, all of the users on the network must have similar software or hardware. Although it is convenient for a business partner to connect to another company's network via VPN, access to specific applications and files should be limited to certain authorized users versus all users on a VPN.[26] Firewalls, intrusion detection software and authorization tools can be used to secure valuable data (Section 21.14).

For more information about IPSec, visit the *IPSec Developers Forum* at **www.ip-sec.com**. Also, check out the Web site for the *IPSec Working Group* of the IETF at **www.ietf.org/html.charters/ipsec-charter.html**.

21.11 Authentication

As we discussed throughout the chapter, authentication is one of the fundamental requirements for e-business and m-business security. In this section, we will discuss some of the technologies used to authenticate users in a network, such as *Kerberos*, *biometrics* and *sin-*

gle sign-on. We conclue the section with a discussion of Microsoft Passport—a technology that combines several methods of authentication.

21.11.1 Kerberos

Firewalls do not protect users from internal security threats to their local area network. Internal attacks are common and can be extremely damaging. For example, disgruntled employees with network access can wreak havoc on an organization's network or steal valuable proprietary information. It is estimated that 70 percent to 90 percent of attacks on corporate networks are internal.[27] *Kerberos* is a freely available, open-source protocol developed at MIT. It employs secret-key cryptography to authenticate users in a network and to maintain the integrity and privacy of network communications.

Authentication in a Kerberos system is handled by a main Kerberos system and a secondary *Ticket Granting Service* (*TGS*). This system is similar to the key distribution centers described in Section 23.3. The main Kerberos system authenticates a client's identity to the TGS; the TGS authenticates client's rights to access specific network services.

Each client in the network shares a secret key with the Kerberos system. This secret key may be used by multiple TGSs in the Kerberos system. The client starts by entering a login name and password into the Kerberos authentication server. The authentication server maintains a database of all clients in the network. The authentication server returns a *Ticket-Granting Ticket* (*TGT*) encrypted with the client's secret key that it shares with the authentication server. Since the secret key is known only by the authentication server and the client, only the client can decrypt the TGT, thus authenticating the client's identity. Next, the client's system sends the decrypted TGT to the Ticket Granting Service to request a *service ticket*. The service ticket authorizes the client's access to specific network services. Service tickets have a set expiration time. Tickets may be renewed by the TGS.

21.11.2 Biometrics

An innovation in security is likely to be *biometrics*. Biometrics uses unique personal information, such as fingerprints, eyeball iris scans or face scans, to identify a user. This system eliminates the need for passwords, which are much easier to steal. Have you ever written down your passwords on a piece of paper and put the paper in your desk drawer or wallet? These days, people have passwords and PIN codes for everything—Web sites, networks, e-mail, ATM cards and even for their cars. Managing all of those codes can become a burden. Recently, the cost of biometrics devices has dropped significantly. Keyboard-mounted fingerprint scanning, face scanning and eye scanning devices are being used in place of passwords to log into systems, check e-mail or access secure information over a network. Each user's iris scan, face scan or fingerprint is stored in a secure database. Each time a user logs in, his or her scan is compared with the database. If a match is made, the login is successful. Two companies that specialize in biometrics devices are IriScan (**`www.iriscan.com`**) and Keytronic (**`www.keytronic.com`**). For additional resources, see Section 21.16.

Currently, passwords are the predominant means of authentication; however, we are beginning to see a shift to smart cards and Biometrics. Microsoft recently announced that it will include the *Biometric Application Programming Interface* (*BAPI*) in future versions of Windows, which will make it possible for companies to integrate biometrics into their

systems.[28] *Two-factor authentication* uses two means to authenticate the user, such as biometrics or a smart card used in combination with a password. Though this system could potentially be compromised, using two methods of authentication is more secure than just using passwords alone.

Keyware Inc. has already implemented a wireless biometrics system that stores user voiceprints on a central server. Keyware also created *layered biometric verification* (*LBV*), which uses multiple physical measurements—face, finger and voice prints—simultaneously. The LBV feature enables a wireless biometrics system to combine biometrics with other authentication methods, such as PIN and PKI.[29]

Identix Inc. also provides biometrics authentication technology for wireless transactions. The Identix fingerprint scanning device is embedded in handheld devices. The Identix service offers *transaction management* and *content protection* services. Transaction management services prove that transactions took place, and content protection services control access to electronic documents, including limiting a user's ability to download or copy documents.[30]

Wireless biometrics is not widely used at this point. Fingerprint scanners must be accompanied by fingerprint readers installed in mobile devices. Wireless device manufacturers are hesitant to build in fingerprint readers because the technology is expensive. Laptops have begun to accommodate biometric security, but cell phones are slower to advance due to limited memory and processing power.[31]

One of the major concerns with biometrics is the issue of privacy. Implementing fingerprint scanners means that organizations will be keeping databases with each employee's fingerprint. Do people want to provide their employers with such personal information? What if that data is compromised? To date, most organizations that have implemented biometrics systems have received little, if any, resistance from employees.

21.11.3 Single Sign-On

To access multiple applications on different servers, users must provide a separate password for authentication on each. Remembering multiple passwords is cumbersome. People tend to write their passwords down, creating security threats.

S*ingle sign-on* systems allow users to login once with a single password. Users can access multiple applications. It is important to secure single sign-on passwords, because if the password becomes available to hackers, all applications can be accessed and attacked.

There are three types of single sign-on services: *workstation logon scripts*, *authentication server scripts* and *tokens*. Workstation logon scripts are the simplest form of single sign-on. Users login at their workstations, then choose applications from a menu. The workstation logon script sends the user's password to the application servers, and the user is authenticated for future access to those applications. Workstation logon scripts do not provide a sufficient amount of security since user passwords are stored on the PC in plaintext. Anyone who can access the workstation can take the user's password. Authentication server scripts authenticate users with a central server. The central server controls connections between the user and the applications the user wishes to access. Authentication server scripts are more secure than workstation logon scripts because passwords are kept on the server, which is more secure than the individual PC.

The most advanced single sign-on systems use token-based authentication. Once a user is authenticated, a non-reusable token is issued to the user to access specific applications.

The logon for creating the token is secured with encryption or with a single password, which is the only password the user needs to remember or change. The only problem with token authentication is that all applications must be built to accept tokens instead of traditional logon passwords.[32]

21.11.4 Microsoft® Passport

Microsoft Passport incorporates authentication, online purchasing, single-sign on and several other technologies that we have discussed into one product that can be used over several different sites over the Internet. Passport users only need to sign in once, with the main Passport authentication server, and they will be recognized as a unique user at each of the Passport-enabled sites they visit. With this technology, users can sign in with a main server, and from that point, check their e-mail, chat with friends and make purchases online—without entering a password for each application.

Once a user logs in, the Passport provides authentication information to the participating sites, but the actual Passport password is safe with the secured database. Passport uses SSL to send the username, password and digital wallet data to the central server. **[***<memberservices.passport.com/HELP/MSRV_HELP_howsecure.asp>***]** The authentication information that sites receive is in the form of a digital key. This key is unique to each user and is verifiable by the Passport database (similar to the PKI architecture). To provide for a greater level of security, the Passport database and the sites that adapt this technology refresh the keys that are used in authentication. The less time that a key is in circulation, the less time an attacker has to analyze the key or use a compromised key. Microsoft Passport also provides for protection from brute-force cracking. If an attacker enters a certain number of incorrect passwords at the log in prompt, Passport temporarily suspends the account for several minutes. This action prevents brute-force programs from repeatedly trying passwords until finding the correct one.

Cookies on the user's computer store profile information after it has been encrypted. When a user logs out of the Passport, all of the personal information that was stored in the cookies is deleted.

Microsoft incorporates Passport technology into many of its upcoming products. Windows XP, the .NET framework and Hailstorm are based on Microsoft Passport.

For more information on Microsoft Passport (and to sign up for a free Passport account), visit **`www.passport.com`**.

21.12 Security Attacks

Recent cyberattacks on e-businesses have made the front pages of newspapers worldwide. *Denial-of-service attacks* (*DoS*), *viruses* and *worms* have cost companies billions of dollars. In this section, we will discuss the different types of attacks and the steps you can take to protect your information.

21.12.1 Denial-of-Service (DoS) Attacks

A denial-of-service attack occurs when a system is forced to behave improperly. In many DoS attacks, a network's resources are taken up by unauthorized traffic, restricting the access of legitimate users. Typically, the attack is performed by flooding servers with data

packets. Denial-of-service attacks usually require the power of a network of computers working simultaneously, although some skillful attacks can be achieved with a single machine. Denial-of-service attacks can cause networked computers to crash or disconnect, disrupting service on a Web site or even shutting down critical systems such as telecommunications or flight-control centers

e-Fact 21.6

Approximately 4,000 sites experience denial-of-service every week.[33, 34]

Another type of denial-of-service attack targets the *routing tables* of a network. Routing tables are the road map of a network, providing directions for data to get from one computer to another. This type of attack is accomplished by modifying the routing tables, thus disabling network activity. For example, the routing tables can be changed to send all data to one address in the network.

In a *distributed denial-of-service attack*, the packet flooding does not come from a single source, but from many separate computers. Actually, such an attack is rarely the concerted work of many individuals. Instead, it is the work of a single individual who has installed viruses on various computers, gaining illegitimate use of the computers to carry out the attack. Distributed denial-of-service attacks can be difficult to stop, since it is not clear which requests on a network are from legitimate users and which are part of the attack. In addition, it is particularly difficult to catch the culprit of such attacks, because the attacks are not carried out directly from the attacker's computer.

Who is responsible for viruses and denial-of-service attacks? Most often the responsible parties are referred to as *hackers* or *crackers*. Hackers and crackers are usually skilled programmers. According to some, hackers break into systems just for the thrill of it, without causing any harm to the compromised systems (except, perhaps, humbling and humiliating their owners). Either way, hackers break the law by accessing or damaging private information and computers. Crackers have malicious intent and are usually interested in breaking into a system to shut down services or steal data. In February 2000, distributed denial-of-service attacks shut down a number of high-traffic Web sites, including Yahoo!, eBay, CNN Interactive and Amazon. In this case, a cracker used a network of computers to flood the Web sites with traffic that overwhelmed the sites' computers. Although denial-of-service attacks merely shut off access to a Web site and do not affect the victim's data, they can be extremely costly. For example, when eBay's Web site went down for a 24-hour period on August 6, 1999, its stock value declined dramatically.[35]

21.12.2 Viruses and Worms

Viruses are pieces of code—often sent as attachments or hidden in audio clips, video clips and games—that attach to, or overwrite other programs to replicate themselves. Viruses can corrupt files or even wipe out a hard drive. Before the Internet was invented, viruses spread through files and programs (such as video games) transferred to computers by removable disks. Today, viruses are spread over a network simply by sharing “infected” files embedded in e-mail attachments, documents or programs. A worm is similar to a virus, except that it can spread and infect files on its own over a network; worms do not need to be attached to another program to spread. Once a virus or worm is released, it can spread rapidly, often infecting millions of computers worldwide within minutes or hours.

There are many classes of computer viruses. A *transient virus* attaches itself to a specific computer program. The virus is activated when the program is run and deactivated when the program is terminated. A more powerful type of virus is a *resident virus*, which, once loaded into the memory of a computer, operates for the duration of the computer's use. Another type of virus is the *logic bomb*, which triggers when a given condition is met, such as a *time bomb* that is activated when the clock on the computer matches a certain time or date.

A *Trojan horse* is a malicious program that hides within a friendly program or simulates the identity of a legitimate program or feature, while actually causing damage to the computer or network in the background. The Trojan horse gets its name from the story of the Trojan War in Greek history. In this story, Greek warriors hid inside a wooden horse, which the Trojans took within the walls of the city of Troy. When night fell and the Trojans were asleep, the Greek warriors came out of the horse and opened the gates to the city, letting the Greek army enter the gates and destroy the city of Troy. Trojan horse programs can be particularly difficult to detect, since they appear to be legitimate and useful applications. Also commonly associated with Trojan horses are *backdoor programs*, which are usually resident viruses that give the sender complete, undetected access to the victim's computer resources. These types of viruses are especially threatening to the victim, as they can be set up to log every keystroke (capturing all passwords, credit card numbers, etc.) No matter how secure the connection between a PC supplying private information and the server receiving the information, if a backdoor program is running on a computer, the data is intercepted before any encryption is implemented. In June 2000, news spread of a Trojan horse virus disguised as a video clip sent as an e-mail attachment. The Trojan horse virus was designed to give the attacker access to infected computers, potentially to launch a denial-of-service attack against Web sites.[36]

Two of the most famous viruses to date are *Melissa*, which struck in March 1999, and the *ILOVEYOU virus* that hit in May 2000. Both viruses cost organizations and individuals billions of dollars. The Melissa virus spread in Microsoft Word documents sent via e-mail. When the document was opened, the virus was triggered. Melissa accessed the Microsoft Outlook address book on that computer and automatically sent the infected Word attachment by e-mail to the first 50 people in the address book. Each time another person opened the attachment, the virus would send out another 50 messages. Once in a system, the virus infected any subsequently saved files.

The ILOVEYOU virus was sent as an attachment to an e-mail posing as a love letter. The message in the e-mail said "Kindly check the attached love letter coming from me." Once opened, the virus accessed the Microsoft Outlook address book and sent out messages to the addresses listed, helping to spread the virus rapidly worldwide. The virus corrupted all types of files, including system files. Networks at companies and government organizations worldwide were shut down for days trying to remedy the problem and contain the virus. This virus accentuated the importance of scanning file attachments for security threats before opening them.

e-Fact 21.7

Estimates for damage caused by the ILOVEYOU virus were as high as $10 billion to $15 billion, with the majority of the damage done in just a few hours.

Why do these viruses spread so quickly? One reason is that many people are too willing to open executable files from unknown sources. Have you ever opened an audio clip or video clip from a friend? Have you ever forwarded that clip to other friends? Do you know who created the clip and if any viruses are embedded in it? Did you open the ILOVE YOU file to see what the love letter said?

Most antivirus software is reactive, going after viruses once they are discovered, rather than protecting against unknown viruses. New antivirus software, such as Finjan Software's SurfinGuard® (**www.finjan.com**), looks for executable files attached to e-mail and runs the executables in a secure area to test if they attempt to access and harm files. For more information about antivirus software, see the **McAfee.com**: Antivirus Utilities feature.

***McAfee.com*: Antivirus Utilities**

McAfee.com provides a variety of antivirus utilities (and other utilities) for users whose computers are not continuously connected to a network, for users whose computers are continuously connected to a network (such as the Internet) and for users connected to a network via wireless devices, such as personal digital assistants.

For computers that are not continuously connected to a network, McAfee provides its antivirus software *VirusScan*®. This software is configurable to scan files for viruses on demand or to scan continuously in the background as the user does his or her work.

For computers that are network and Internet accessible, McAfee provides its online **McAfee.com** Clinic. Users with a subscription to McAfee Clinic can use the online virus software from any computer they happen to be using. As with VirusScan software on stand-alone computers, users can scan their files on demand. A major benefit of the Clinic is its *ActiveShield* software. Once installed, ActiveShield can be configured to scan every file that is used on the computer or just the program files. It can also be configured to check automatically for virus definition updates and notify the user when such updates become available. The user simply clicks on the supplied hyperlink in an update notification to connect to the Clinic site and clicks on another hyperlink to download the update. Thus, users can keep their computers protected with the most up-to-date virus definitions at all times, an important factor in protection from viruses.

McAfee.com *VirusScan Wireless* provides virus protection for Palm™ handhelds, Pocket PC and other handheld devices. VirusScan Wireless is installed on the user's PC. Each time the user syncs the handheld device, the software scans for viruses. If a virus is detected, the sync is terminated until the user deletes the virus. For more information about McAfee, visit **www.mcafee.com**. Also, check out Norton security products from Symantec, at **www.symantec.com**. Symantec is a leading security software vendor. Its product Norton™ Internet Security 2000 provides protection against hackers, viruses and threats to privacy for both small businesses and individuals.

21.12.3 Software Exploitation, Web Defacing and Cybercrime

Another problem plaguing e-businesses is *software exploitation* by hackers. In addition to constantly updating virus and firewall programs, every program on a networked machine should be checked for vulnerabilities. However, with millions of software products available and more vulnerabilities discovered daily, this becomes an enormous task. One common vulnerability exploitation method is a *buffer overflow*, in which a program is overwhelmed by an input of more data than it has allocated space for. Buffer overflow attacks can cause systems to crash or, more dangerously, allow arbitrary code to be run on a machine. *BugTraq* was created in 1993 to list vulnerabilities, how to exploit them and how to repair them. For more information about BugTraq, visit **`www.securityfocus.com`**.

Web defacing is another popular form of attack, wherein the crackers illegally enter an organization's Web site and change the contents. CNN Interactive has issued a special report titled "Insurgency on the Internet," with news stories about hackers and their online attacks. Included is a gallery of defaced sites. One notable case of Web defacing occurred in 1996, when Swedish crackers changed the Central Intelligence Agency Web site (**`www.odci.gov/cia`**) to read "Central Stupidity Agency." The vandals put obscenities, political messages, notes to system administrators and links to adult-content sites on the page. Many other popular and large Web sites have been defaced. Defacing Web sites has become overwhelmingly popular amongst crackers today, causing archives of affected sites (with records of more than 15,000 vandalized sites) to close because of the volume in which sites were being vandalized daily.[37]

Cybercrime can have significant financial implications on an organization.[38] Companies need to protect their data, intellectual property, customer information, etc. Implementing a *security policy* is key to protecting an organization's data and network. When developing a security plan, organizations must assess their vulnerabilities and the possible threats to security. What information do they need to protect? Who are the possible attackers and what is their intent—data theft or damaging the network? How will the organization respond to incidents?[39] For more information about security and security plans, visit **`www.cerias.com`** and **`www.sans.org`**. Visit **`www.baselinesoft.com`** to check out books and CD-ROMs on security policies. Baseline Software's book, *Information Policies Made Easy: Version 7* includes over 1000 security policies. This book is used by numerous Fortune 200 companies.

e-Fact 21.8

According to the GartnerGroup, 70% of computer crime is committed by disgruntled employees.[40]

The rise in cybercrimes has prompted the U. S. government to take action. Under the National Information Infrastructure Protection Act of 1996, denial-of-service attacks and distribution of viruses are federal crimes punishable by fines and jail time. For more information about the U. S. government's efforts against cybercrime or to read about recently prosecuted cases, visit the U.S. Department of Justice Web site, at **`www.usdoj.gov/criminal/cybercrime/compcrime.html`**. Also check out **`www.cybercrime.gov`**, a site maintained by the Criminal Division of the U. S. Department of Justice.

The *CERT®* (*Computer Emergency Response Team*) *Coordination Center* at Carnegie Mellon University's Software Engineering Institute responds to reports of viruses and

denial-of-service attacks and provides information on network security, including how to determine if a system has been compromised. The site provides detailed incident reports of viruses and denial-of-service attacks, including descriptions of the incidents, their impact and solutions. The site also includes reports of vulnerabilities in popular operating systems and software packages. The *CERT Security Improvement Modules* are excellent tutorials on network security. These modules describe the issues and technologies used to solve network security problems. For more information, visit the CERT Web site, at **www.cert.org**.

To learn more about how you can protect yourself or your network from hacker attacks, visit AntiOnline™, at **www.antionline.com**. This site has security-related news and information, a tutorial titled "Fight-back! Against Hackers," information about hackers and an archive of hacked sites. You can find additional information about denial-of-service attacks and how to protect your site at **www.irchelp.org/irchelp/nuke**.

21.13 Running Resticted Python Code

Python code is platform independent, so, once Python code is written it can be run virtually anywhere. Many programmers access Python code remotely by downloading it and running it using a Python interpreter installed on the local system. This method raises security issues, however— once the code runs on a local machine it could gain unauthorized access to local files or otherwise misuse the machine.

One solution to prevent executing damaging code is to run code in a *restricted environment*. Restricted environment is a virtual machine that provides access only to the resources the program may need that are physically available on the local machine. If the code is unable to access sensitive resources (such as a hard drive or network) it will not be able to damage such resources.

21.13.1 Module rexec

Module **rexec** contains the **RExec** class used to execute Python code in a restricted environment. An **RExec** instance supports several methods that perform restricted execution, such as **r_eval**. Code that executes in this environment has limited access to modules and built-in Python functions—the programmer has complete control over the environment in which the code runs. A default restricted environment imports several modules, including **__builtins__** and **sys**. **RExec** only can restrict access to some resources such as a disk or network but it cannot limit the amount of memory or CPU time used.

21.13.2 Module Bastion

Module **Bastion** can be used to restrict access to specific objects, rather than the entire environment. The **Bastion** object wraps an object and controls the access to this object. **Bastion** provides precise control over the methods of the object, achieved by supplying a filter function when creating a **Bastion** instance. The filter function takes a method name as an argument and returns true if that method can be accessed. By default, methods of the object are not accessible.

When code tries to access a restricted method, an **AttributeError** exception is thrown. This happens because, from the code's point of view, the method does not exist. In the restricted environment, this method was never defined and thus it is not accessible.

21.13.3 Web browser example

Figure 21.7 demonstrates a modified version of the Web browser we presented in Chapter 20 (Fig. 20.1). The modified browser checks whether the requested page ends with the **.py** extension. If so, the browser runs the Python code in a restricted environment.

```
# Fig. 21.7: fig20_02.py
# This program displays the contents of a file on a Web server.

from Tkinter import *
import tkMessageBox
import Pmw
import urllib
import urlparse
import Bastion
import rexec

class WebBrowser( Frame ):
   """A simple Web browser"""

   def __init__( self ):
      """Create the Web browser GUI"""

      Frame.__init__( self )
      Pmw.initialise()
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Simple Web Browser" )
      self.master.geometry( "400x300" )

      self.address = Entry( self )
      self.address.pack( fill = X, padx = 5, pady = 5 )
      self.address.bind( "<Return>", self.getPage )

      self.contents = Pmw.ScrolledText( self,
         text_state = DISABLED )
      self.contents.pack( expand = YES, fill = BOTH, padx = 5,
         pady = 5 )

      # create restricted environment
      self.restricted = rexec.RExec()
      self.module = self.restricted.add_module( "__main__" )
      self.environment = self.module.__dict__

      # add browser to environment
      self.environment[ "browser" ] = Bastion.Bastion( self )

   def setColor( self, color ):
```

Fig. 21.7 Web browser example.

```
      """Set browser's background color"""

      self.configure( background = color )

   def _setColor( self, color ):
      """Set browser's background"""

      self.configure( background = color )

   def setText( self, text ):
      """Set the text of the ScrolledText component"""

      self.contents.settext( text )

   def runCode( self, statement ):
      """Run a Python statement in restricted environment"""

      try:
         self.restricted.r_exec( statement ) # execute in rexec
      except AttributeError, name:
         tkMessageBox.showerror( "Error",
            "Restricted code tried to access forbidden " + \
            "attribute:" + str( name ) )

   def getPage( self, event ):
      """Parse the URL and addressing scheme and retrieve file"""

      # parse the URL
      myURL = event.widget.get()
      components = urlparse.urlparse( myURL )
      self.contents.text_state = NORMAL

      # if addressing scheme not specified, use http
      if components[ 0 ] == "":
         myURL = "http://" + myURL

      # connect and retrieve the file
      try:
         tempFile = urllib.urlopen( myURL ).read()
      except IOError:
         self.contents.settext( "Error finding file" )
      else:
         tempFile = tempFile.replace( "\r\n", "\n" )

         if myURL[-3:] == ".py":
            self.runCode( tempFile )
         else:
            self.contents.settext( tempFile ) # show results

      self.contents.text_state = DISABLED

def main():
   WebBrowser().mainloop()
```

Fig. 21.7 Web browser example.

```

if __name__ == "__main__":
    main()
```

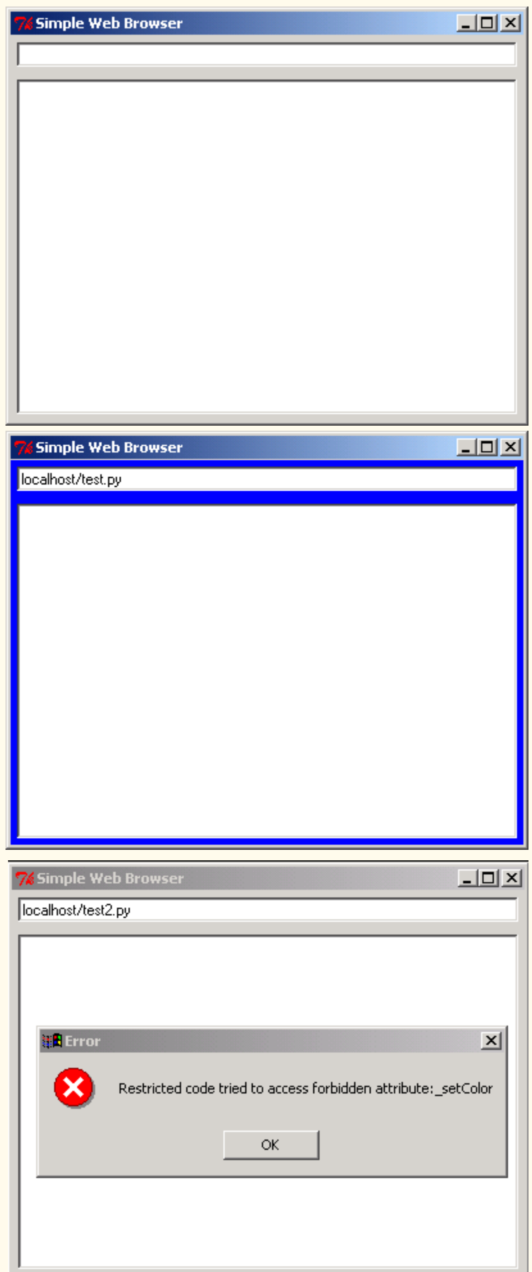


Fig. 21.7 Web browser example.

Line 34 creates an instance of class **RExec**. Line 35 gets the environment's **__main__** module of that environment. The instance defines an environment that contains a list of accessible modules and built-in functions (e.g. **raw_input** or **abs**). It has its own environment, including a list of accessible modules and built in methods. Method **add_module** adds a new module to the list of the modules allowed in the restricted environment and returns a reference to that module. If the environment already permits access to the module, method **add_module** simply returns a reference to the specified module. Method **add_module** does not import the module into the restricted environment; the method only modifies the list of modules that the restricted code may import.

Line 34 gets the reference to the dictionary **__dict__** that contains the module-global bindings for the restricted environment. A **Bastion** module wraps a Web browser component and adds it to the module-global namespace of the restricted environment (line 39). The restricted code now may access and manipulate the Web browser component. By wrapping the Web browser component with class **Bastion**, we allow the program to control how the restricted code accesses the browser. By default, code many not access a **Bastion** instance's data member or any methods that begin with the underscore (_) letter. The code may access method that do not begin with the underscore character.

To demonstrate code execution, lines 41–49 add two methods to the **WebBrowser**. Both **setColor** (lines 41–44) and **_setColor** (lines 46–49) set the foreground color of the **WebBrowser**. By default, code may not access a Bastion-wrapped browser object's **_setColor** method.

The screenshots in Fig. 21.7 demonstrate the result of running the code in Fig. 21.8 and Fig. 21.9. The first screenshot is the browser in its original state. The second screenshot is the result of running the code in Fig. 21.8. The browser has changed its background color to blue. The final screenshot demonstrates what happens when the code in Fig. 21.7 attempts to change color using restricted **_setColor**.

```
browser.setColor( "blue" )
```

Fig. 21.8

```
browser._setColor( "red" )
```

Fig. 21.9

21.14 Network Security

The goal of network security is to allow authorized users access to information and services, while preventing unauthorized users from gaining access to, and possibly corrupting, the network. There is a trade-off between network security and network performance: Increased security often decreases the efficiency of the network.

In this section, we will discuss the various aspects of network security. We will discuss firewalls, which keep unauthorized users out of the network, and authorization servers, which allow users to access specific applications based on a set of pre-defined criteria. We will then look at intrusion detection systems that actively monitor a network for intrusions and attacks.

21.14.1 Firewalls

A basic tool in network security is the *firewall*. The purpose of a firewall is to protect a *local area network* (*LAN*) from intruders outside the network. For example, most companies have internal networks that allow employees to share files and access company information. Each LAN can be connected to the Internet through a gateway, which usually includes a firewall. For years, one of the biggest threats to security came from employees inside the firewall. Now that businesses rely heavily on access to the Internet, an increasing number of security threats are originating outside the firewall—from the hundreds of millions of people connected to the company network by the Internet.[51] A firewall acts as a safety barrier for data flowing into and out of the LAN. Firewalls can prohibit all data flow not expressly allowed, or can allow all data flow that is not expressly prohibited. The choice between these two models is up to the network security administrator and should be based on the need for security versus the need for functionality.

There are two main types of firewalls: *packet-filtering firewalls* and *application-level gateways*. A packet-filtering firewall examines all data sent from outside the LAN and rejects any data packets that have local network addresses. For example, if a hacker from outside the network obtains the address of a computer inside the network and tries to sneak a harmful data packet through the firewall, the packet-filtering firewall will reject the data packet, since it has an internal address, but originated from outside the network. A problem with packet-filtering firewalls is that they consider only the source of data packets; they do not examine the actual data. As a result, malicious viruses can be installed on an authorized user's computer, giving the hacker access to the network without the authorized user's knowledge. The goal of an application-level gateway is to screen the actual data. If the message is deemed safe, then the message is sent through to the intended receiver.

Using a firewall is probably the most effective and easiest way to add security to a small network.[52] Often, small companies or home users who are connected to the Internet through permanent connections, such as DSL lines, do not employ strong security measures. As a result, their computers are prime targets for crackers to use in denial-of-service attacks or to steal information. It is important for all computers connected to the Internet to have some degree of security for their systems. Numerous firewall software products are available. Several products are listed in the Web resources in Section 6.15.

Air gap technology is a network security solution that complements the firewall. It secures private data from external traffic accessing the internal network. The *air gap* separates the internal network from the external network, and the organization decides which information will be made available to external users. *Whale Communications* created the *e-Gap System*, which is composed of two computer servers and a *memory bank*. The memory bank does not run an operating system, therefore hackers cannot take advantage of common operating system weaknesses to access network information.

Air gap technology does not allow outside users to view the network's structure, preventing hackers from searching the layout for weak spots or specific data. The e-Gap *Web Shuttle* feature allows safe external access by restricting the system's *back office*, which is where an organization's most sensitive information and IT-based business processes are controlled. Users who want to access a network hide behind the air gap, where the authentication server is located. Authorized users gain access through a *single sign-on* capability, allowing them to use one log-in password to access authorized areas of the network.

The e-Gap *Secure File Shuttle* feature moves files in and out of the network. Each file is inspected behind the air gap. If the file is deemed safe, it is carried by the File Shuttle into the network.[53]

Air gap technology is used by e-commerce organizations to allow their clients and partners to access information automatically, thus reducing the cost of inventory management. Military, aerospace and government industries, which store highly sensitive information, use air gap technology.

SANS Institute: Security Research and Education

The *System Administration, Networking and Security Institute* (*SANS*), founded in 1989, is a security research and education organization with over 96,000 members (**www.sans.org**). SANS sells security training, certification programs and publications. The organization also offers several free, publicly-available services such as security alerts and news.

Each year, SANS publishes the *Roadmap to Security Tools and Services Poster*—a resource that includes information about key security technologies, lists of security vendors that specialize in each technology and URLs with additional security information. The poster also includes directions on how to order approximately 20 white papers. To order a copy of the poster and to request copies of the technical white papers, go to **www.sans.org/tools.htm**.

The SANS Information Security Reading Room is an excellent resource for security information. The site has hundreds of articles and case studies organized by security topic. Topics include authentication, attacking attackers, intrusion detection, securing code, standards and many more. For more information, visit **www.sans.org/infosecFAQ/index.htm**.

SANS offers three free newsletters. *SANS NewsBites* is a free weekly e-mail newsletter that lists key security news articles with a short summary of each article and a link to the complete resource. Go to **www.sans.org/newlook/digests/newsbites.htm** to view the latest newsletter, to view past newsletters or to subscribe. *Security Alert Consensus* (*SAC*) is a weekly summary of new security alerts and countermeasures. Subscribers can opt to receive information on specific operating systems based on their particular needs. The *SANS Windows Security Digest* lists Windows NT security updates, threats and bugs. To subscribe to any of the SANS e-mail newsletters, go to **www.sans.org/sansnews**.

The SANS *Global Incident Analysis Center* (*GIAC*) records current attacks and analyzes each attack. Network and systems administrators can use this information to help them defend their networks and systems against attacks. Reports are made readily available to the public at **www.sans.org/giac.htm** and **www.incidents.org**.

21.14.2 Intrusion Detection Systems

What happens if a hacker gets inside your firewall? How do you know if an intruder has penetrated the firewall? Also, how do you know if unauthorized employees are accessing

restricted applications? *Intrusion detection systems* monitor networks and application *log files*—files containing information on files, including who accessed them and when—so if an intruder makes it into the network or an unauthorized application, the system detects the intrusion, halts the session and sets off an alarm to notify the system administrator.[54]

Host-based intrusion detection systems monitor system and application log files. They can be used to scan for Trojan horses, for example. *Network-based intrusion detection* software monitors traffic on a network for any unusual patterns that might indicate DoS attacks or attempted entry into a network by an unauthorized user. Companies can then check their log files to determine if indeed there was an intrusion and if so, they can attempt to track the offender. Check out the intrusion detection products from Cisco (**www.cisco.com/warp/public/cc/pd/sqsw/sqidsz**), Hewlett-Packard (**www.hp.com/security/home.html**) and Symantec (**www.symantec.com**).

The *OCTAVE*[SM] (*Operationally Critical Threat, Asset and Vulnerability Evaluation*) *method*, under development at the Software Engineering Institute at Carnegie Mellon University, is a process for evaluating security threats of a system. There are three phases in OCTAVE: building threat profiles, identifying vulnerabilities, and developing security solutions and plans. In the first stage, the organization identifies its important information and assets, then evaluates the levels of security required to protect them. In the second phase, the system is examined for weaknesses that could compromise the valuable data. The third phase is to develop a security strategy as advised by an analysis team of three to five security experts assigned by OCTAVE. This approach is one of the firsts of its kind, in which the owners of computer systems not only get to have professionals analyze their systems, but also participate in prioritizing the protection of crucial information.[55]

21.15 Steganography

Steganography is the practice of hiding information within other information. The term literally means "covered writing." Like cryptography, steganography has been used since ancient times. Steganography allows you to take a piece of information, such as a message or image, and hide it within another image, message or even an audio clip. Steganography takes advantage of insignificant space in digital files, in images or on removable disks.[56] Consider a simple example: If you have a message that you want to send secretly, you can hide the information within another message, so that no one but the intended receiver can read it. For example, if you want to tell your stockbroker to buy a stock and your message must be transmitted over an unsecure channel, you could send the message "BURIED UNDER YARD." If you have agreed in advance that your message is hidden in the first letters of each word, the stock broker picks these letters off and sees "BUY."

An increasingly popular application of steganography is *digital watermarks* for intellectual property protection. An example of a conventional watermark is shown in Fig. 21.10. A digital watermark can be either visible or invisible. It is usually a company logo, copyright notification or other mark or message that indicates the owner of the document. The owner of a document could show the hidden watermark in a court of law, for example, to prove that the watermarked item was stolen.

Digital watermarking could have a substantial impact on e-commerce. Consider the music industry. Music publishers are concerned that MP3 technology is allowing people to distribute illegal copies of songs and albums. As a result, many publishers are hesitant to

put content online, as digital content is easy to copy. Also, since CD-ROMs are digital, people are able to upload their music and share it over the Web. Using digital watermarks, music publishers can make indistinguishable changes to a part of a song at a frequency that is not audible to humans, to show that the song was, in fact, copied. Microsoft Research is developing a watermarking system for digital audio, which would be included with default Windows media players. In this digital watermarking system, data such as licensing information is embedded into a song; the media player will not play files with invalid information.

e-Fact 21.9

Record Companies are losing approximately $5 billion per year due to piracy. [57]

Blue Spike's Giovanni™ digital watermarking software uses cryptographic keys to generate and embed steganographic digital watermarks into digital music and images (Fig. 7.8). The watermarks can be used as proof of ownership to help digital publishers protect their copyrighted material. The watermarks are undetectable by anyone who is not privy to the embedding scheme, and thus the watermarks cannot be identified and removed. The watermarks are placed randomly.

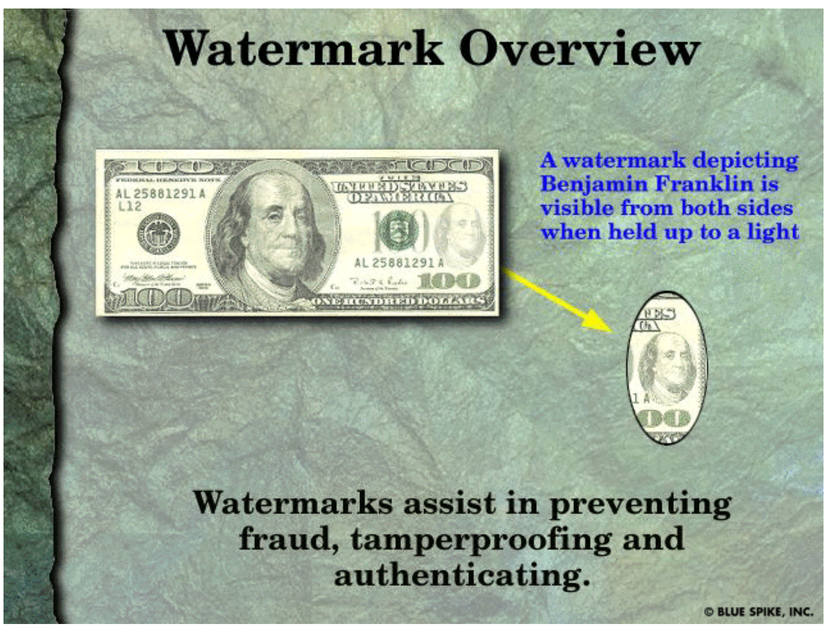


Fig. 21.10 Example of a conventional watermark. (Courtesy of Blue Spike, Inc.)

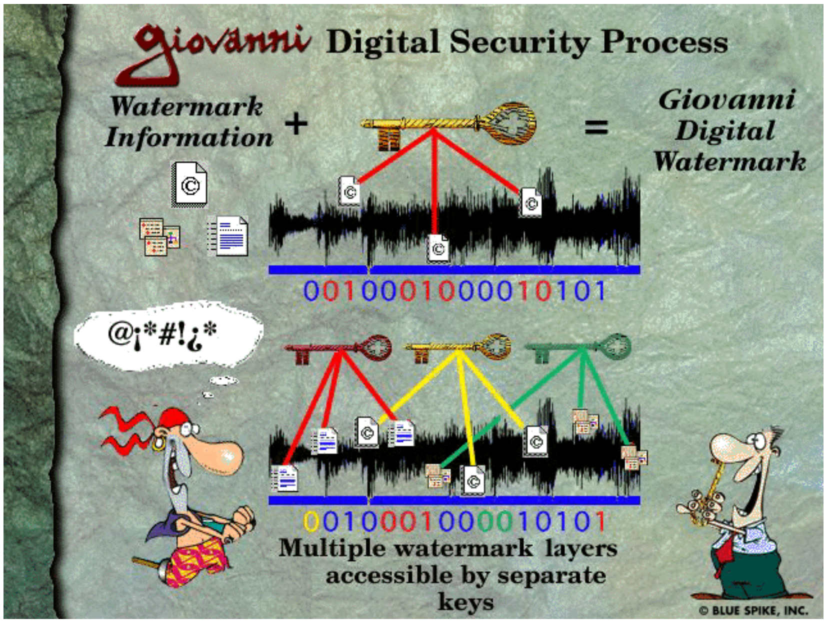


Fig. 21.11 An example of steganography: Blue Spike's Giovanni digital watermarking process. (Courtesy of Blue Spike, Inc.)

Giovanni incorporates cryptography and steganography. It generates a secret key based on an encryption algorithm and the contents of the audio or image file. The key is then used to place (and eventually decode) the watermark in the file. The software identifies the perceptually insignificant areas of the image or audio file, enabling a digital watermark to be embedded inaudibly, invisibly and in such a way that if the watermark is removed, the content is likely to be damaged.

Digital watermarking capabilities are built into some image-editing software applications, such as Adobe PhotoShop 5.5 (**www.adobe.com**). Companies that offer digital watermarking solutions include Digimarc (**www.digimark.com**) and Cognicity (**www.cognicity.com**).

In the last few chapters, we discussed the technologies involved in building and running an m-business, and how to secure online and wireless transactions and communications. In Chapter 7, Legal, Ethical and Social Issues; Web Accessibility, we discuss a number of major legal and ethical concerns that have developed from the introduction of the Internet and the World Wide Web.

21.16 Internet and World Wide Web Resources

Security Resource Sites

www.securitysearch.com
This is a comprehensive resource for computer security. The site has thousands of links to products, security companies, tools and more. The site also offers a free weekly newsletter with information about vulnerabilities.

www.esecurityonline.com
This site is a great resource for information on online security. The site has links to news, tools, events, training and other valuable security information and resources.

theory.lcs.mit.edu/~rivest/crypto-security.html
The *Ronald L. Rivest: Cryptography and Security* site has an extensive list of links to security resources, including newsgroups, government agencies, FAQs, tutorials and more.

www.w3.org/Security/Overview.html
The *W3C Security Resources* site has FAQs, information about W3C security and e-commerce initiatives and links to other security related Web sites.

web.mit.edu/network/ietf/sa
The Internet Engineering Task Force (IETF), which is an organization concerned with the architecture of the Internet, has working groups dedicated to Internet Security. Visit the *IETF Security Area* to learn about the working groups, join the mailing list or check out the latest drafts of the IETF's work.

dir.yahoo.com/Computers_and_Internet/Security_and_Encryption
The *Yahoo Security and Encryption* page is a great resource for links to Web sites security and encryption.

www.counterpane.com/hotlist.html
The Counterpane Internet Security, Inc., site includes links to downloads, source code, FAQs, tutorials, alert groups, news and more.

www.rsasecurity.com/rsalabs/faq
This site is an excellent set of FAQs about cryptography from RSA Laboratories, one of the leading makers of public key cryptosystems.

www.nsi.org/compsec.html
Visit the National Security Institute's *Security Resource Net* for the latest security alerts, government standards, and legislation, as well as security FAQs links and other helpful resources.

www.itaa.org/infosec
The Information Technology Association of America (ITAA) *InfoSec* site has information about the latest U.S. government legislation related to information security.

staff.washington.edu/dittrich/misc/ddos
The *Distributed Denial of Service Attacks* site has links to news articles, tools, advisory organizations and even a section on security humor.

www.infoworld.com/cgi-bin/displayNew.pl?/security/links/security_corner.htm
The Security Watch site on **Infoword.com** has loads of links to security resources.

www.antionline.com
AntiOnline has security-related news and information, a tutorial titled "Fight-back! Against Hackers," information about hackers and an archive of hacked sites.

www.microsoft.com/security/default.asp
The Microsoft security site has links to downloads, security bulletins and tutorials.

www.grc.com
This site offers a service to test the security of your computer's Internet connection.

www.sans.org/giac.html
Sans Institute presents information on system and security updates, along with new research and discoveries. The site offers current publications, projects, and weekly digests.

www.pactetstorm.securify.com
The Packet Storm page describes the twenty latest advisories, tools, and exploits. This site also provides links to the top security news stories.

www.xforce.iss.net
This site allows one to search a virus by name, reported date, expected risk, or affected platforms. Updated news reports can be found on this page.

www.ntbugtraq.com
This site provides a list and description of various Windows NT Security Exploits/Bugs encountered by Windows NT users. One can download updated service applications.

nsi.org/compsec.html
The Security Resource Net page states various warnings, threats, legislation and documents of viruses and security in an organized outline.

www.securitystats.com
This computer security site provides statistics on viruses, web defacements and security spending.

Magazines, Newsletters and News sites

www.networkcomputing.com/consensus
The *Security Alert Consensus* is a free weekly newsletter with information about security threats, holes, solutions and more.

www.atstake.com/security_news
Visit this site for daily security news.

www.infosecuritymag.com
Information Security Magazine has the latest Web security news and vendor information.

www.issl.org/cipher.html
Cipher is an electronic newsletter on security and privacy from the Institute of Electrical and Electronics Engineers (IEEE). You can view current and past issues online.

securityportal.com
The *Security Portal* has news and information about security, cryptography and the latest viruses.

www.scmagazine.com
SC Magazine has news, product reviews and a conference schedule for security events.

www.cnn.com/TECH/specials/hackers
Insurgency on the Internet from CNN Interactive has news on hacking, plus a gallery of hacked sites.

Government Sites for Computer Security

www.cit.nih.gov/security.html
This site has links to security organizations, security resources and tutorials on PKI, SSL and other protocols.

cs-www.ncsl.nist.gov
The *Computer Security Resource Clearing House* is a resource for network administrators and others concerned with security. This site has links to incident-reporting centers, information about security standards, events, publications and other resources.

www.cdt.org/crypto
Visit the Center for Democracy and Technology for U. S. legislation and policy news regarding cryptography.

www.epm.ornl.gov/~dunigan/security.html
This site has links to loads of security-related sites. The links are organized by subject and include resources on digital signatures, PKI, smart cards, viruses, commercial providers, intrusion detection and several other topics.

www.alw.nih.gov/Security
The *Computer Security Information* page is an excellent resource, providing links to news, newsgroups, organizations, software, FAQs and an extensive number of Web links.

www.fedcirc.gov
The Federal Computer Incident Response Capability deals with the security of government and civilian agencies. This site has information about incident statistics, advisories, tools, patches and more.

axion.physics.ubc.ca/pgp.html
This site has a list of freely available cryptosystems, along with a discussion of each system and links to FAQs and tutorials.

www.ifccfbi.gov
The Internet Fraud Complaint Center, founded by the Justice Department and the FBI, fields reports of Internet fraud.

www.disa.mil/infosec/iaweb/default.html
The Defense Information Systems Agency's *Information Assurance* page includes links to sites on vulnerability warnings, virus information and incident-reporting instructions, as well as other helpful links.

www.nswc.navy.mil/ISSEC/
The objective of this site is to provide information on protecting your computer systems from security hazards. Contains a page on hoax versus real viruses.

www.cit.nih.gov/security.html
You can report security issues at this site. The site also lists official federal security policies, regulations, and guidelines.

cs-www.ncsl.nist.gov/
The Computer Security Resource Center provides services for vendors and end users. The site includes information on security testing, management, technology, education and applications.

Advanced Encryption Standard (AES)

csrc.nist.gov/encryption/aes
The official site for the AES includes press releases and a discussion forum.

www.esat.kuleuven.ac.be/~rijmen/rijndael/
Visit this site for information about the Rijndael algorithm.

home.ecn.ab.ca/~jsavard/crypto/co040801.htm
This AES site includes an explanation of the algorithm with helpful diagrams and examples.

Internet Security Vendors

www.rsasecurity.com
RSA is one of the leaders in electronic security. Visit its site for more information about its current products and tools, which are used by companies worldwide.

www.ca.com/protection
Computer Associates is a vendor of Internet security software. It has various software packages to help companies set up a firewall, scan files for viruses and protect against viruses.

www.checkpoint.com
Check Point™ Software Technologies Ltd. is a leading provider of Internet security products and services.

www.opsec.com
The Open Platform for Security (OPSEC) has over 200 partners that develop security products and solutions using the OPSEC to allow for interoperability and increased security over a network.

www.baltimore.com
Baltimore Security is an e-commerce security solutions provider. Their UniCERT digital certificate product is used in PKI applications.

www.ncipher.com
nCipher is a vendor of hardware and software products, including an SSL accelerator that increases the speed of secure Web server transactions and a secure key management system.

www.entrust.com
Entrust Technologies provides e-security products and services.

www.antivirus.com
ScanMail® is an e-mail virus detection program for Microsoft Exchange.

www.zixmail.com
Zixmail™ is a secure e-mail product that allows you to encrypt and digitally sign your messages using different e-mail programs.

web.mit.edu/network/pgp.html
Visit this site to download *Pretty Good Privacy*® freeware. PGP allows you to send messages and files securely.

www.certicom.com
Certicom provides security solutions for the wireless Internet.

www.raytheon.com
Raytheon Corporation's *SilentRunner* monitors activity on a network to find internal threats, such as data theft or fraud.

SSL

developer.netscape.com/tech/security/ssl/protocol.html
This Netscape page has a brief description of SSL, plus links to an SSL tutorial and FAQs.

www.netscape.com/security/index.html
The Netscape Security Center is an extensive resource for Internet and Web security. You will find news, tutorials, products and services on this site.

psych.psy.uq.oz.au/~ftp/Crypto
This FAQs page has an extensive list of questions and answers about SSL technology.

www.visa.com/nt/ecomm/security/main.html
Visa International's security page includes information on SSL and SET. The page includes a demonstration of an online shopping transaction, which explains how SET works.

www.openssl.org
The *Open SSL Project* provides a free, open source toolkit for SSL.

Public-key Cryptography

www.entrust.com
Entrust produces effective security software products using Public Key Infrastructure (PKI).

www.cse.dnd.ca
The Communication Security Establishment has a short tutorial on Public Key Infrastructure (PKI) that defines PKI, public-key cryptography and digital signatures.

www.magnet.state.ma.us/itd/legal/pki.htm
The Commonwealth of Massachusetts Information Technology page has loads of links to sites related to PKI that contain information about standards, vendors, trade groups and government organizations.

www.ftech.net/~monark/crypto/index.htm
The Beginner's Guide to Cryptography is an online tutorial and includes links to other sites on privacy and cryptography.

www.faqs.org/faqs/cryptography-faq
The *Cryptography FAQ* has an extensive list of questions and answers.

www.pkiforum.org
The PKI Forum promotes the use of PKI.

www.counterpane.com/pki-risks.html
Visit the Counterpane Internet Security, Inc.'s site to read the article "Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure."

Digital Signatures

www.ietf.org/html.charters/xmldsig-charter.html
The *XML Digital Signatures* site was created by a group working to develop digital signatures using XML. You can view the group's goals and drafts of their work.

www.elock.com
E-Lock Technologies is a vendor of digital-signature products used in Public Key Infrastructure. This site has an FAQs list covering cryptography, keys, certificates and signatures.

www.digsigtrust.com
The Digital Signature Trust Co. is a vendor of Digital Signature and Public Key Infrastructure products. It has a tutorial titled "Digital Signatures and Public Key Infrastructure (PKI) 101."

Digital Certificates

www.verisign.com
VeriSign creates digital IDs for individuals, small businesses and large corporations. Check out its Web site for product information, news and downloads.

www.thawte.com
Thawte Digital Certificate Services offers SSL, developer and personal certificates.

www.silanis.com/index.htm
Silanis Technology is a vendor of digital-certificate software.

www.belsign.be
Belsign issues digital certificates in Europe. It is the European authority for digital certificates.

www.certco.com
Certco issues digital certificates to financial institutions.

www.openca.org
Set up your own CA using open-source software from The OpenCA Project.

Digital Wallets

www.globeset.com
GlobeSet is a vendor of digital-wallet software. Its site has an animated tutorial demonstrating the use of an electronic wallet in an SET transaction.

www.trintech.com
Trintech digital wallets handle SSL and SET transactions.

wallet.yahoo.com
The *Yahoo! Wallet* is a digital wallet that can be used at thousands of Yahoo! Stores worldwide.

Firewalls

www.interhack.net/pubs/fwfaq
This site provides an extensive list of FAQs on firewalls.

www.spirit.com/cgi-bin/report.pl
Visit this site to compare firewall software from a variety of vendors.

www.zeuros.co.uk/generic/resource/firewall
Zeuros is a complete resource for information about firewalls. You will find FAQs, books, articles, training and magazines on this site.

www.thegild.com/firewall
The *Firewall Product Overview* site has an extensive list of firewall products, with links to each vendor's site.

csrc.ncsl.nist.gov/nistpubs/800-10
Check out this firewall tutorial from the U.S. Department of Commerce.

www.watchguard.com
WatchGuard® Technologies, Inc., provides firewalls and other security solutions for medium to large organizations.

Kerberos

www.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html
This site is an extensive list of FAQs on Kerberos from the Naval Research Laboratory.

web.mit.edu/kerberos/www
Kerberos: The Network Authentication Protocol is a list of FAQs provided by MIT.

www.contrib.andrew.cmu.edu/~shadow/kerberos.html
The Kerberos Reference Page has links to several informational sites, technical sites and other helpful resources.

www.pdc.kth.se/kth-krb
Visit this site to download various Kerberos white papers and documentation.

Biometrics

www.iosoftware.com/products/integration/fiu500/index.htm
This site describes a security device that scans a user's fingerprint to verify identity.

www.identix.com/flash_index.html
Identix specializes in fingerprinting systems for law enforcement, access control and network security. Using its fingerprint scanners, you can log on to your system, encrypt and decrypt files and lock applications.

www.iriscan.com
Iriscan's *PR Iris*™ can be used for e-commerce, network and information security. The scanner takes an image of the user's eye for authentication.

www.keytronic.com
Key Tronic manufactures keyboards with fingerprint recognition systems.

IPSec and VPNs

www.checkpoint.com
Check Point™ offers combined firewall and VPN solutions. Visit their resource library for links to numerous white papers, industry groups, mailing lists and other security and VPN resources.

www.ietf.org/html.charters/ipsec-charter.html
The IPSec Working Group of the Internet Engineering Task Force (IETF) is a resource for technical information related to the IPSec protocol.

www.icsalabs.com/html/communities/ipsec/certification/certified_products/index.shtml
Visit this site for a list of certified IPSec products, plus links to an IPSec glossary and other related resources.

www.ip-sec.com
The IPSec Developers Forum allows vendors and users to test the interoperability of different IPSec products. The site includes technical documents related to the IPSec protocol.

www.vpnc.org
The Virtual Private Network Consortium, which has VPN standards, white papers, definitions and archives. VPNC also offers compatibility testing with current VPN standards.

Steganography and Digital Watermarking

www.bluespike.com/giovanni/giovmain.html
Blue Spike's *Giovanni* watermarks help publishers of digital content protect their copyrighted material and track their content that is distributed electronically.

www.outguess.org
Outguess is a freely available steganographic tool.

www.cl.cam.ac.uk/~fapp2/steganography/index.html
The Information Hiding Homepage has technical information, news and links related to digital watermarking and steganography.

www.demcom.com
DemCom's *Steganos Security Suite* software allows you to encrypt and hide files within audio, video, text or HTML files.

www.cognicity.com
Cognicity specializes in digital-watermarking solutions for the music and entertainment industries.

Newsgroups

news:comp.security.firewalls

news:comp.security.unix

news:comp.security.misc

news:comp.protocols.kerberos

TERMINOLOGY

128-bit IV
3DES
ActiveShield
Advanced Encryption Standard (AES)
application-level gateway
assemblies
asymmetric algorithm
authentication
authentication header (AH)
availability
backdoor program
binary string
biometrics
bit
block
block cipher
brute-force cracking
buffer overflow
BugTraq
Caesar cipher
CERT (Computer Emergency Response Team)
CERT Security Improvement Modules
certificate authority (CA)
certificate authority hierarchy
certificate repository
certificate revocation list (CRL)
cipher
ciphertext
collision
contact interface
contactless interface
content protection
CPU
cracker
cryptanalysis
cryptanalytic attack
cryptography
cryptosystem
Data Encryption Standard (DES)
data packet
decryption
denial-of-service (DoS) attack
denial-of-service attack
DES cracker machine
Diffie-Hellman Key Agreement Protocol
digital certificate
digital envelope
digital ID
digital signature
Digital Signature Algorithm (DSA)

digital watermarking
distributed denial-of-service attack
Dynamic Proxy Navigation (DPN)
electronic shopping cart
Elliptic Curve Cryptography (ECC)
Encapsulating Security Payload (ESP)
encryption
Enhanced Security Network (ESN)
firewall
gateway
GSM (Global System for Mobile Communications)
hacker
hash function
hash value
identity permissions
ILOVEYOU Virus
initialization vector (IV)
integrity
integrity check (IC)
Internet Engineering Task Force (IETF)
Internet Key Exchange (IKE)
Internet Policy Registration Authority (IPRA)
Internet Protocol (IP)
Internet Security, Applications, Authentication and Cryptography (ISAAC)
IP address
IP spoofing
IPSec (Internet Protocol Security)
IV collision
Kerberos
key
key agreement protocol
key distribution center
key generation
key length
key management
layered biometric verification (LBV)
Liberty Trojan horse
Lightweight Extensible Authentication Protocol (LEAP)
local area network (LAN)
logic bomb
Lucifer
man-in-the-middle attack
masquerading
MD5 hashing algorithm
Melissa Virus
memory card
message digest
message integrity
microprocessor card
Microsoft Authenticode
Microsoft Intermediate Language (MSIL)
Microsoft Passport
mobile code
Mobile Wireless Internet Forum
Mobiletrust certificate authority
National Institute of Standards and Technology (NIST)
network security
nonrepudiation
Online Certificate Status Protocol (OCSP)
packet
packet-filtering firewall
PCI (peripheral component interconnect) card

permissions
personal identification number (PIN)
plaintext
point-to-point connection
policy creation authority
Pretty Good Privacy (PGP)
privacy
private key
protocol
public key
Public Key Infrastructure (PKI)
public-key algorithms
public-key cryptography
resident virus
restricted algorithms
Rijndael
role based access control (RBAC)
root certificate authority
root key
routing table
RSA encryption memory
RSA Security, Inc.
secret key
Secure Enterprise Proxy
Secure Sockets Layer (SSL)
security policy file
service ticket
session key
single sign-on
smart card
socket
software exploit
steganography
substitution cipher
symmetric encryption algorithm
TCP/IP (Transmission Control Protocol/Internet Protocol)
Ticket Granting Service (TGS)
Ticket Granting Ticket (TGT)
time bomb
timestamping
timestamping agency
transaction management
transient virus
transposition cipher
Triple DES
Trojan horse virus
Trustpoint
VeriSign
Virtual Private Network (VPN)
virus
Web defacing
Wide area network (WAN)
worm

SELF-REVIEW EXERCISES

21.1 State whether the following are *true* or *false*. If the answer is *false*, explain why.
a) In a public-key algorithm, one key is used for both encryption and decryption.
b) Digital certificates are intended to be used indefinitely.
c) Secure Sockets Layer protects data stored on a merchant's server.

d) Digital signatures can be used to provide undeniable proof of the author of a document.
e) In a network of 10 users communicating using public-key cryptography, only 10 keys are needed in total.
f) The security of modern cryptosystems lies in the secrecy of the algorithm.
g) Increasing the security of a network often decreases its functionality and efficiency.
h) Firewalls are the single most effective way to add security to a small computer network.
i) Kerberos is an authentication protocol that is used over TCP/IP networks.
j) SSL can be used to connect a network of computers over the Internet.
k) Hacker attacks, such as Denial-of-Service and viruses, can cause e-business to lose billions of dollars.

21.2 Fill in the blanks in each of the following statements:
a) Cryptographic algorithms in which the message's sender and receiver both hold an identical key are called __________.
b) A __________ is used to authenticate the sender of a document.
c) In a __________, a document is encrypted using a secret key and sent with that secret key, encrypted using a public-key algorithm.
d) A certificate that needs to be revoked before its expiration date is placed on a __________.
e) The recent wave of network attacks that have hit companies such as eBay, and Yahoo are known as __________.
f) A digital fingerprint of a document can be created using a __________.
g) The four main issues addressed by cryptography are __________, __________, __________ and __________.
h) A customer can store purchase information and data on multiple credit cards in an electronic purchasing and storage device called a __________.
i) Trying to decrypt ciphertext without knowing the decryption key is known as __________.
j) A barrier between a small network and the outside world is called a __________.
k) A hacker that tries every possible solution to crack a code is using a method known as __________.

ANSWERS TO SELF-REVIEW EXERCISES

21.1 a) False. The encryption key is different from the decryption key. One is made public, and the other is kept private. b) False. Digital certificates are created with an expiration date to encourage users to change their public/private-key pair periodically. c) False. Secure Sockets Layer is an Internet security protocol, which secures the transfer of information in electronic communication. It does not protect data stored on a merchant's server. d) False. A user who digitally signed a document could later intentionally give up his or her private key and then claim that the document was written by an imposter. Thus, timestamping a document is necessary, so that users cannot repudiate documents written before the pubic/private-key pair is reported as invalidated. e) False. Each user needs a public key and a private key. Thus, in a network of 10 users, 20 keys are needed in total. f) False. The security of modern cryptosystems lies in the secrecy of the encryption and decryption keys. g) True. h) True. i) True. j) False, IPSec can connect a whole network of computers, while SSL can only connect two secure systems. k) True.

21.2 a) symmetric key algorithms. b) digital signature. c) digital envelope. d) certificate revocation list. e) distributed denial-of-service attacks. f) hash function. g) privacy, authentication, integrity, non-repudiation. h) electronic wallet. i) cryptanalysis. j) firewall. k) brute-force hacking.

EXERCISES

21.3 What can online businesses do to prevent hacker attacks, such as denial-of-service attacks and virus attacks?

21.4 Define the following security terms:

a) digital signature
b) hash function
c) symmetric key encryption
d) digital certificate
e) denial-of-service attack
f) worm
g) message digest
h) collision
i) triple DES
j) session keys

21.5 Define each of the following security terms, and give an example of how it is used:

a) secret-key cryptography
b) public-key cryptography
c) digital signature
d) digital certificate
e) hash function
f) SSL
g) Kerberos
h) firewall

21.6 Write the full name and describe each of the following acronyms:

a) PKI
b) IPSec
c) CRL
d) AES
e) SSL

21.7 List the four problems addressed by cryptography, and give a real-world example of each.

21.8 Compare symmetric-key algorithms with public-key algorithms. What are the benefits and drawbacks of each type of algorithm? How are these differences manifested in the real-world uses of the two types of algorithms?

WORKS CITED

1. A. Harrison, "Xerox Unit Farms Out Security in $20M Deal," Computerworld 5 June 2000: 24.

2. "What the Experts are Saying About Security: Facts and Quotes," from an OKENA company Press kit.

3. "RSA Laboratories' Frequently Asked Questions About Today's Cryptography, Version 4.1," 2000 **<www.rsasecurity.com/rsalabs/faq>**.

4. **<www-math.cudenver.edu/~wcherowi/courses/m5410/m5410des.html>**

5. M. Dworkin, "Advanced Encryption Standard (AES) Fact Sheet," 5 March 2001.

6. **<www.esat.kuleuven.ac.be/~rijmen/rijndael>**

7. **<www.rsasecurity.com/rsalabs/rsa_algorithm>**

8. **<www.pgpi.org/doc/overview>**

9. **<www.rsasecurity.com/rsalabs/faq>**.

10. **<userpages.umbc.edu/~mabzug1/cs/md5/md5.html>**.

11. T. Russell, "The Cyptographic Landscape for PKI Smart Cards," *Internet Security Advisor* March/April 2001: 22.

12. G. Hulme, "VeriSign Gave Microsoft Certificates to Imposter," *Information Week* 3 March 2001.

13. R. Yasin, "PKI Rollout to Get Cheaper, Quicker," *InternetWeek* 24 July 2000: 28.

14. C. Ellison and B. Schneier, "Ten Risks of PKI: What You're not Being Told about Public Key Infrastructure," *Computer Security Journal* 2000.

15. "What's So Smart About Smart Cards?" *Smart Card Forum*.

16. T. Russell, "The Cyptographic Landscape for PKI Smart Cards," *Internet Security Advisor*, March/April 2001: 22.

17. S. Abbot, "The Debate for Secure E-Commerce," *Performance Computing* February 1999: 37-42.

18. T. Wilson, "E-Biz Bucks Lost Under the SSL Train," *Internet Week* 24 May 1999: 1, 3.

19. H. Gilbert, "Introduction to TCP/IP," 2 February 1995 **<www.yale.edu/pclt/COMM/TCPIP.HTM>**.

20. RSA Laboratories, "Security Protocols Overview," 1999 **<www.rsasecurity.com/standards/protocols>**.

21. M. Bull, "Ensuring End-to-End Security with SSL," *Network World* 15 May 2000: 63.

22. **<www.cisco.com/warp/public/44/solutions/network/vpn.shtml>**.

23. S. Burnett and S. Paine, *RSA Security's Official Guide to Cryptography* (Berkeley: Osborne/McGraw-Hill, 2001) 210.

24. D. Naik, *Internet Standards and Protocols* Microsoft Press 1998: 79-80.

25. M. Grayson, "End the PDA Security Dilemma," *Communication News* February 2001: 38-40.

26. T. Wilson, "VPNs Don't Fly Outside Firewalls," *Internet Week*, 28 May 2001.

27. S. Gaudin, "The Enemy Within," *Network World* 8 May 2000: 122-126.

28. D. Deckmyn, "Companies Push New Approaches to Authentication," *Computerworld* 15 May 2000: 6.

29. "Centralized Authentication," **<www.keyware.com>**.

30. J. Vijayan, "Biometrics Meet Wireless Internet," *Computerworld* 17 July 2000: 14.

31. C. Nobel, "Biometrics Targeted For Wireless Devices," *eweek* 31 July 2000: 22.

32. F. Trickey, "Secure Single Sign-On: Fantasy or Reality," *CSI* **<www.gocsi.com>**

33. D. Moore, G. Voelker and S. Savage, "Inferring Internet Denial-of-Service Activity."

34. J. Schwartz, "Computer Vandals Clog Antivandalism Web Site," *The New York Times* 24 May 2001.

35. "Securing B2B," *Global Technology Business* July 2000: 50-51.

36. H. Bray, "Trojan Horse Attacks Computers, Disguised as a Video Chip," *The Boston Globe* 10 June 2000: C1+.

37. T.Bridis, "U.S. Archive of Hacker Attacks To Close Because It Is Too Busy," *The Wall Street Journal* 24 May 2001: B10.

38. R. Marshland, "Hidden Cost of Technology," *Financial Times* 2 June 2000: 5.

39. F. Avolio, "Best Practices in Network Security," *Network Computing* 20 March 2000: 60-72.

40. "Industry Statistics," from an AbsoluteSoftware company Press kit.

41. J. Singer, R. Fink, "A Security Analysis of C#"

42. **<msdn.microsoft.com/library/default.asp?url=/library/en-us/dnc-sspec/html/vclrfcsharpspec_a.asp>**

43. J. Singer, R. Fink, "A Security Analysis of C#"

44. **<msdn.microsoft.com/msdnmag/issues/01/02/CAS/CAS.asp>**

45. **<msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemSecurityPermissionsFileIOPermissionClass-Topic.asp>**

46. **<www.msdn.microsoft.com/library/dotnet/cpguide/cpconpermis-sions.html>**

47. **<msdn.microsoft.com/msdnmag/issues/01/02/CAS/CAS.asp>**

48. **<msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemsecuritycodeaccesspermissionmember-stopic.asp>**

49. ***R. Yasin, "Security First for Visa", InternetWeek, 13 November 2000.***

50. **L. Lorek, "E-Commerce Insecurity", Interactive Week, April 23, 2001.**

51. R. Marshland, 5.

52. T. Spangler, "Home Is Where the Hack Is," *Inter@ctive Week* 10 April 2000: 28-34.

53. "Air Gap Technology," *Whale Communications* **<www.whale-com.com>**.

54. O. Azim and P. Kolwalkar, "Network Intrusion Monitoring," *Advisor.com/Security* March/April 2001: 16-19.

55. "OCTAVE Information Security Risk Evaluation," 30 January 2001 **<www.cert.org/octave/methodintro.html>**.

56. S. Katzenbeisser and F. Petitcolas, *Information Hiding: Techniques for Steganography and Digital Watermarking* (Norwood: Artech House, Inc., 2000) 1-2.

57. D.McCullagh, "MS May Have File-Trading Answer," 1 May 2001 **<www.wired.com/news/print/0,1294,43389,00.html>**.

[***Notes To Reviewers***]

- Please pay close attention to Sections 21.8 and 21.13—the Python-specific sections.
- We will post this chapter (with solutions to exercises) for second-round review.
- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send us e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **cheryl.yaeger@deitel.net**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
- Please be constructive. This book will be published soon. We all want to publish the best possible book.
- If you find something that is incorrect, please show us how to correct it.
- Please read all the back matter including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

22

Data Structures

Objectives

- To be able to form linked data structures using self-referential classes and recursion.
- To be able to create and manipulate dynamic data structures such as linked lists, queues, stacks and binary trees.
- To understand various important applications of linked data structures.
- To understand how to create reusable data structures with inheritance and composition.

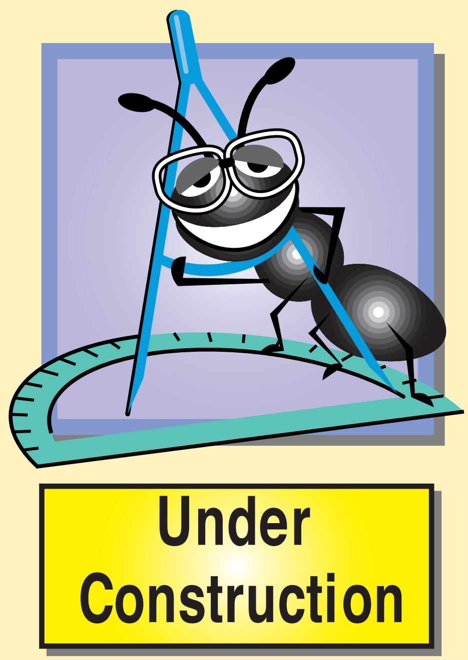
Much that I bound, I could not free;
Much that I freed returned to me.
Lee Wilson Dodd

'Will you walk a little faster?' said a whiting to a snail,
'There's a porpoise close behind us, and he's treading on my tail.'
Lewis Carroll

There is always room at the top.
Daniel Webster

Push on — keep moving.
Thomas Morton

I think that I shall never see
A poem lovely as a tree.
Joyce Kilmer



Outline

22.1 Introduction

We have studied Python's high-level data types such as lists, tuples and dictionaries. This chapter introduces the general topic of *data structures* that underlies Python's basic data types. *Linked lists* are collections of data items "lined up in a row"—insertions and removals are made anywhere in a linked list. *Stacks* are important in compilers and operating systems—insertions and removals are made only at one end of a stack—its *top*. *Queues* represent waiting lines*;* insertions are made at the back (also referred to as the *tail*) of a queue, and removals are made from the front (also referred to as the *head*) of a queue. *Binary trees* facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories and compiling expressions into machine language. These data structures have many other interesting applications.

We will discuss the major types of data structures and implement programs that create and manipulate these data structures. We use classes and inheritance to create and package these data structures for reusability and maintainability.

Although basic Python lists can serve as stacks and queues, studying this chapter and creating these structures "from scratch" is solid preparation for higher-level computer science courses. The chapter examples are practical programs that you will be able to use in more advanced courses and in industry applications. The exercises include a rich collection of useful applications.

22.2 Self-Referential Classes

A *self-referential class* contains a reference member that refers to an instance of the same class type. Consider a class **`Node`** that has two data members—member **`data`** and reference member **`nextNode`**. Member **`nextNode`** refers to an instance of class **`Node`**—an instance of the same class as the one being declared here, hence the term "self-referential class." Member **`nextNode`** is referred to as a *link*—i.e., **`nextNode`** can be used to "tie" an instance of class **`Node`** to another instance of the same type. Class **`Node`** also has five methods: a constructor that receives a value to initialize member **`data`**, a **`setData`** method to set the value of member **`data`**, a **`getData`** method to return the value of member

`data`, a **`setNextNode`** method to set the value of member **`nextNode`** and a **`getNextNode`** method to return the value of member **`nextNode`**.

Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees. Figure 22.1 illustrates two self-referential class instances linked together to form a list. Note that a slash—representing a reference to **`None`**—is placed in the link member of the second self-referential class instance to indicate that the link does not refer to another instance. The slash is only for illustration purposes; it does not correspond to the backslash character in Python. A **`None`** reference normally indicates the end of a data structure.

Common Programming Error 22.1

Not setting the link in the last node of a list to **`None`**.

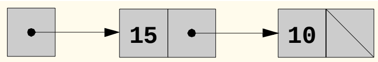


Fig. 22.1 Two self-referential class objects linked together.

22.3 Linked Lists

A *linked list* is a linear collection of self-referential class instances, called *nodes*, connected by reference *links*—hence, the term "linked" list. A linked list is accessed via a reference to the first node of the list. Subsequent nodes are accessed via the reference link stored in each node. By convention, the link in the last node of a list is set to **`None`** to mark the end of the list. Data are stored in a linked list dynamically—each node is created as necessary. A node can contain data of any type, including instances of other classes. Stacks and queues are also linear data structures and, as we will see, are constrained versions of linked lists. Trees are nonlinear data structures.

Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list. Existing list elements do not need to be moved.

Performance Tip 22.1

Insertion and deletion in a regular sorted list can be time-consuming—all the elements following the inserted or deleted element must be shifted appropriately. However, insertion and deletion in a sorted linked list requires only three changes to reference links (at most).

Linked list nodes are normally not stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous. Figure 22.2 illustrates a linked list with several nodes.

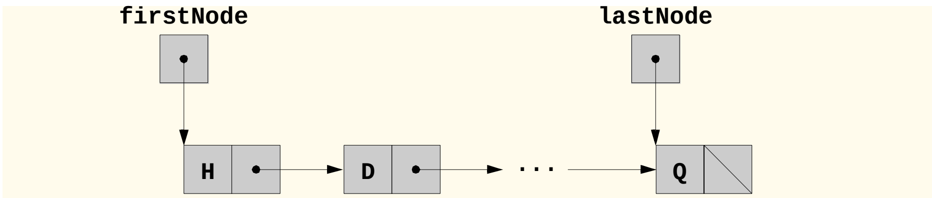


Fig. 22.2 A graphical representation of a list.

The program of Figure 22.3 uses a **`List`** instance to manipulate a list of integer values. The driver program (**`fig15_02.py`**) provides five options:

1. Insert a value at the beginning of the list (method **`insertAtFront`**).
2. Insert a value at the end of the list (method **`insertAtBack`**).
3. Delete a value from the front of the list (method **`removeFromFront`**).
4. Delete a value from the end of the list (method **`removeFromBack`**).
5. Terminate the list processing.

A detailed discussion of the program follows.

```
# Fig. 22.3: List.py
# Classes List and Node definition

class Node:
   "Single node in a data structure"

   def __init__( self, data ):
      "Node constructor"

      self.data = data
      self.nextNode = None

   def getData( self ):
      "Get node data"

      return self.data

   def setData( self, data ):
      "Set node data"

      self.data = data

   def getNextNode( self, ):
      "Get reference to next node"

      return self.nextNode

   def setNextNode( self, newNode ):
      "Set reference to next node"
```

Fig. 22.3 Manipulating a linked list—**`List.py`**.

```

      self.nextNode = newNode;

class List:
   "Linked list"

   def __init__( self ):
      "List constructor"

      self.firstNode = None
      self.lastNode = None

   def __str__( self ):
      "Override print statement"

      if self.isEmpty():
         return "The list is empty"

      currentNode = self.firstNode
      string = "The list is: "

      while currentNode is not None:
         string += str( currentNode.getData() ) + " "
         currentNode = currentNode.getNextNode()

      return string

   def insertAtFront( self, value ):
      "Insert node at front of list"

      newNode = Node( value )

      if self.isEmpty():        # List is empty
         self.firstNode = self.lastNode = newNode
      else:                     # List is not empty
         newNode.setNextNode( self.firstNode )
         self.firstNode = newNode

   def insertAtBack( self, value ):
      "Insert node at back of list"

      newNode = Node( value )

      if self.isEmpty():        # List is empty
         self.firstNode = self.lastNode = newNode
      else:                     # List is not empty
         self.lastNode.setNextNode( newNode )
         self.lastNode = newNode

   def removeFromFront( self ):
      "Delete node from front of list"

      if self.isEmpty():   # raise error on empty list
         raise IndexError, "remove from empty list"
```

Fig. 22.3 Manipulating a linked list—**`List.py`**.

```
      firstNodeValue = self.firstNode.getData()

      if self.firstNode is self.lastNode:   # one node in list
         self.firstNode = self.lastNode = None
      else:
         self.firstNode = self.firstNode.getNextNode()

      return firstNodeValue

   def removeFromBack( self ):
      "Delete node from back of list"

      if self.isEmpty():   # raise error on empty list
         raise IndexError, "remove from empty list"

      lastNodeValue = self.lastNode.getData()

      if self.firstNode is self.lastNode:   # one node in list
         self.firstNode = self.lastNode = None
      else:
         currentNode = self.firstNode

         while currentNode.getNextNode() is not self.lastNode:
            currentNode = currentNode.getNextNode()

         currentNode.setNextNode( None )
         self.lastNode = currentNode

      return lastNodeValue

   def isEmpty( self ):
      "Is the list empty?"

      return self.firstNode is None
```

Fig. 22.3 Manipulating a linked list—**List.py**.

```
# Fig. 22.3: fig22_02.py
# Driver to test class List

import sys
from List import List

def instructions():
   "Print instructions for the user"

   print "Enter one of the following:\n", \
         "  1 to insert at beginning of list\n", \
         "  2 to insert at end of list\n", \
         "  3 to delete from beginning of list\n", \
         "  4 to delete from end of list\n", \
         "  5 to end list processing\n"
```

Fig. 22.3 Manipulating a linked list—**fig22_03.py**.

```
listObject = List()

instructions()
choice = raw_input("? ")

while choice != "5":

   if choice == "1":
      listObject.insertAtFront( raw_input( "Enter value: " ) )
      print listObject
   elif choice == "2":
      listObject.insertAtBack( raw_input( "Enter value: " ) )
      print listObject
   elif choice == "3":

      try:
         value = listObject.removeFromFront()
      except IndexError, message:
         print "Failed to remove:", message
      else:
         print value, "removed from list"
         print listObject

   elif choice == "4":

      try:
         value = listObject.removeFromBack()
      except IndexError, message:
         print "Failed to remove:", message
      else:
          print value, "removed from list"
          print listObject

   else:
      print "Invalid choice:", choice

   choice = raw_input("\n? ")

print "End list test\n"
```

Fig. 22.3 Manipulating a linked list—**fig22_03.py**.

```
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing

? 1
Enter value: 1
The list is: 1

? 1
Enter value: 2
The list is: 2 1

? 2
Enter value: 3
The list is: 2 1 3

? 2
Enter value: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test
```

Fig. 22.3 Manipulating a linked list—**fig22_03.py**.

Figure 22.3 consists of two classes—**Node** and **List**. Encapsulated in each **List** object is a linked list of **Node** instances. **Node** member **nextNode** stores a reference to the next **Node** instance in the linked list.

The **List** class consists of members **firstNode** (a reference to the first **Node** in a **List** instance) and **lastNode** (a reference to the last **Node** in a **List** instance). The constructor initializes both links to **None**. The primary methods of the **List** class are **insertAtFront**, **insertAtBack**, **removeFromFront**, and **removeFromBack**.

Method **isEmpty** is called a *predicate method*—it does not alter the **List**; rather, it determines if the **List** is empty (i.e., the reference to the first **Node** of the **List** is **None**). If the **List** is empty, 1 is returned; otherwise, 0 is returned. Method **__str__** displays the **List**'s contents.

Good Programming Practice 22.1

*Assign **None** to the link member of a new node.*

Software Engineering Observation 22.1

*Because of Python reference counting, when no references to a **List** object exist, the **List** is destroyed, and all **Node** instances the **List** referenced are destroyed (assuming there are no other references to them). However, in a language without reference counting or automatic garbage collection (such as C or C++), it is necessary to remove all references to these instances and destroy them manually (by a destructor, for example).*

Over the next several pages, we discuss each of the methods of the **List** class in detail. Method **insertAtFront** places a new node at the front of the list. The method consists of several steps:

1. Create a new **Node** instance and store the reference in variable **newNode**.
2. If the list is empty, then both **firstNode** and **lastNode** are set to **newNode**.
3. If the list is not empty, then the node referenced by **newNode** is threaded into the list by copying **firstNode** to **newNode.nextNode** so that the new node refers to what used to be the first node of the list, and copying **newNode** to **firstNode** so that **firstNode** now refers to the new first node of the list.

Figure 22.4 illustrates method **insertAtFront**. Part a) of the figure shows the list and the new node before the **insertAtFront** operation. The dotted arrows in part b) illustrate the steps 2 and 3 of the **insertAtFront** operation that enable the node containing **12** to become the new list front.

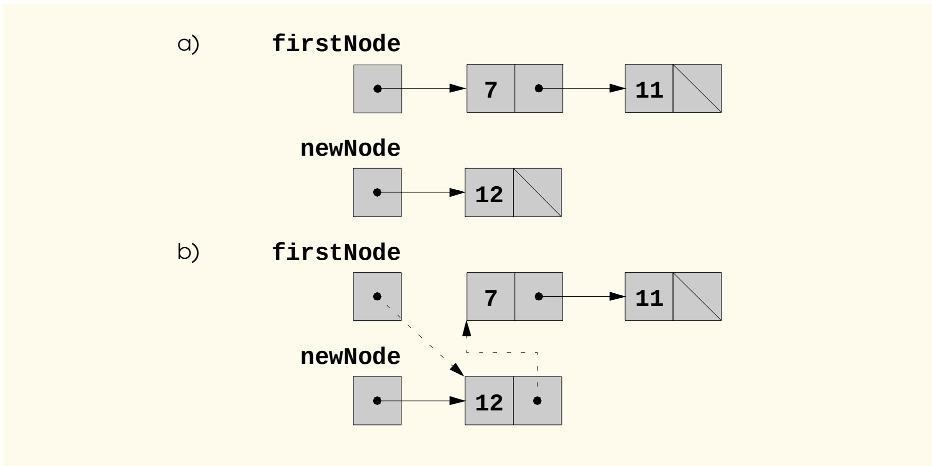


Fig. 22.4 A graphical representation of the **insertAtFront** operation.

Method **insertAtBack** places a new node at the back of the list. The method consists of several steps:

1. Create a new list node that contains **value** and store the node in reference **newNode**.
2. If the list is empty, then both **firstNode** and **lastNode** are set to **newNode**.
3. If the list is not empty, then the node referenced by **newNode** is threaded into the list by copying **newNode** into **lastNode.nextNode** so that the new node is referred to by what used to be the last node of the list, and copying **newNode** to **lastNode** so that **lastNode** now points to the new last node of the list.

Figure 22.5 illustrates an **insertAtBack** operation. Part a) of the figure shows the list and the new node before the operation. The dotted arrows in part b) illustrate the steps of method **insertAtBack** that enable a new node to be added to the end of a list that is not empty.

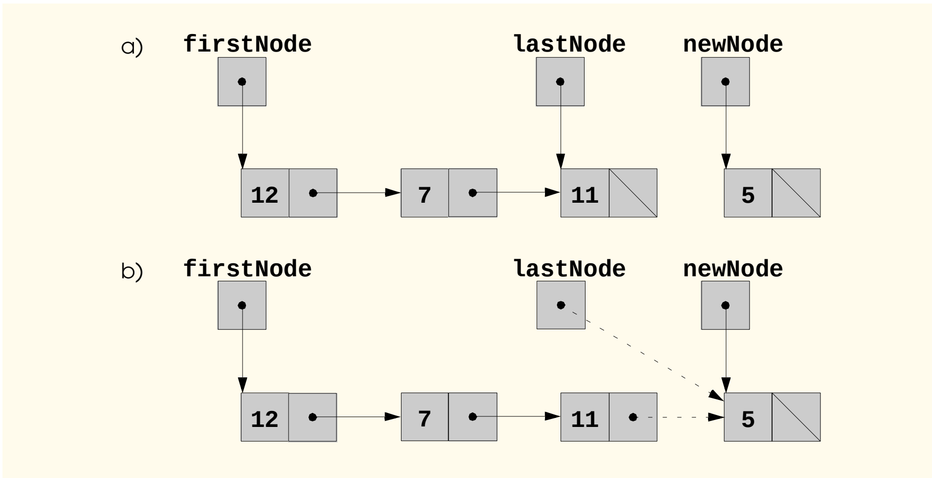


Fig. 22.5 A graphical representation of the **insertAtBack** operation.

Method **removeFromFront** removes the front node of the list and returns the value in that node. The method raises an **IndexError** if an attempt is made to remove a node from an empty list. The method consists of several steps:

1. If the list is empty, raise an **IndexError**.
2. Assign the data from the **firstNode** to variable **firstNodeValue**. The method eventually returns this value.
3. If **firstNode** is equal to **lastNode**, i.e., if the list has only one element prior to the removal attempt, then set **firstNode** and **lastNode** to **None** to de-thread that node from the list (leaving the list empty).
4. If the list has more than one node prior to removal, then leave **lastNode** as is and set **firstNode** to **firstNode.nextNode**, i.e., modify **firstNode** to refer to what was the second node prior to removal (and is the new first node now).
5. After all these reference manipulations are complete, return **firstNodeValue**, the data from the removed node.

Figure 22.6 illustrates method **removeFromFront**. Part a) illustrates the list before the removal operation. Part b) shows actual reference manipulations.

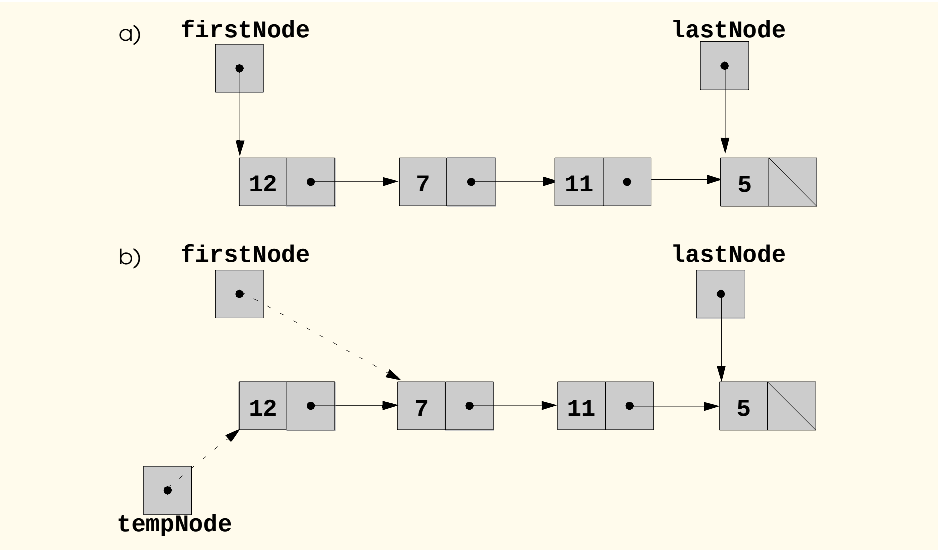


Fig. 22.6 A graphical representation of the **removeFromFront** operation.

Method **removeFromBack** removes the back node of the list and returns the value in that node. The method raises and **IndexError** if an attempt is made to remove a node from an empty list. The method consists of several steps:

1. If the list is empty, raise an **IndexError**.
2. Assign the data from the **lastNode** to variable **lastNodeValue**. The method eventually returns this value.
3. If **firstNode** is equal to **lastNode**, i.e., if the list has only one element prior to the removal attempt, then set **firstNode** and **lastNode** to **None** to de-thread that node from the list (leaving the list empty).
4. If the list has more than one node prior to removal, then assign **currentNode** the node to which **firstNode** refers.
5. Now "walk the list" with **currentNode** until it refers to the node before the last node. This is done with a **while** loop that keeps replacing **currentNode** by **currentNode.nextNode** while **currentNode.nextNode** is not **lastNode**.
6. Set the **nextNode** of **currentNode** to **None** and assign **lastNode** to **currentNode**.
7. After all these reference manipulations are complete, return **lastNodeValue**, the data from the removed node.

Figure 22.7 illustrates method **removeFromBack**. Part a) of the figure illustrates the list before the removal operation. Part b) of the figure shows the actual reference manipulations.

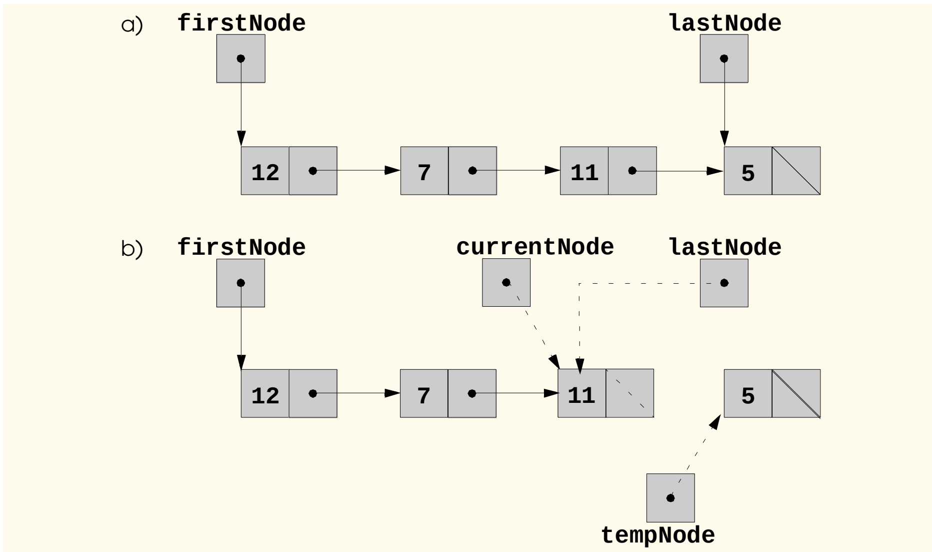


Fig. 22.7 A graphical representation of the **removeFromBack** operation.

Method **__str__** first determines if the list is empty. If so, the method returns **"The list is empty"**. Otherwise, it returns a string that contains each node's data. The method initializes **currentNode** as a copy of **firstNode** and then initializes the string **"The list is: "**. While **currentNode** is not **None**, **currentNode.data** is added to the string and the value of **currentNode.nextNode** is assigned to **currentNode**. Note that if the link in the last node of the list is not **None**, the string creation algorithm will erroneously continue past the end of the list. The string creation algorithm is identical for linked lists, stacks and queues.

The kind of linked list we have been discussing is a *singly linked list*—the list begins with a reference to the first node, and each node contains a reference to the next node "in sequence." This list terminates with a node whose reference member is **None**. A singly linked list may be traversed in only one direction.

A *circular, singly linked list* begins with a reference to the first node, and each node contains a reference to the next node. The "last node" does not contain a reference to **None**; rather, the reference in the last node refers back to the first node, thus closing the "circle."

A *doubly linked list* allows traversals both forwards and backwards. Such a list is often implemented with two "start references"—one that refers to the first element of the list to allow front-to-back traversal of the list, and one that refers to the last element of the list to allow back-to-front traversal of the list. Each node has both a forward reference to the next node in the list in the forward direction and a backward reference to the next node in the list in the backward direction. If the list contains an alphabetized telephone directory, for example, searching for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list. Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

In a *circular, doubly linked list*, the forward reference of the last node refers to the first node, and the backward reference of the first node refers to the last node, thus closing the "circle."

22.4 Stacks

A *stack* is a constrained version of a linked list—new nodes can be added to a stack and removed from a stack only at the top. For this reason, a stack is referred to as a *last-in, first-out (LIFO)* data structure. The link member in the last node of the stack is set to **None** to indicate the bottom of the stack.

Common Programming Error 22.2

Not setting the link in the bottom node of a stack to ***None****.*

The primary methods used to manipulate a stack are ***push*** and ***pop***. Method **push** adds a new node to the top of the stack. Method **pop** removes a node from the top of the stack and returns the popped value to the caller. The method raises an **IndexError** if the stack is empty.

Stacks have many interesting applications. For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks contain the space created for local variables on each invocation of a function. When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's local variables is popped off the stack and those variables are no longer known to the program. Stacks are used by compilers in the process of evaluating expressions and generating machine language code. The exercises explore several applications of stacks.

We will take advantage of the close relationship between lists and stacks to implement a stack class primarily by reusing a list class. We implement the stack class through inheritance of the list class.

The program of Figure 22.8 creates a **Stack** class primarily through inheritance of class **List** of Fig. 22.3 We want the **Stack** to have methods **push** and **pop**. Note that these are essentially the **insertAtFront** and **removeFromFront** methods of class **List**. When we implement the **Stack**'s methods, we then have each of these call the appropriate method of class **List**—**push** calls **insertAtFront**, **pop** calls **removeFromFront**. Of course, class **List** contains other methods (i.e., **insertAtBack** and **removeFromBack**) that we would not use when manipulating instances of class **Stack**. The driver program uses class **Stack** to instantiate a stack instance. Integers 0 through 3 are pushed onto the stack and then popped off the stack.

```
# Fig. 22.8: Stack.py
# Class stack definition

```

Fig. 22.8 Simple stack implementation—**Stack.py**.

```
from List import List

class Stack ( List ):
   "Stack built from linked list"

   def push( self, data ):
      "Push data into stack"

      self.insertAtFront( data )

   def pop( self ):
      "Pop data from stack"

      return self.removeFromFront()
```

Fig. 22.8 Simple stack implementation—**Stack.py**.

```
# Fig. 22.8: fig22_08.py
# Driver to test class Stack

from Stack import Stack

stack = Stack()

print "processing a Stack"

for i in range( 4 ):
   stack.push( i )
   print stack

while not stack.isEmpty():
   pop = stack.pop()
   print pop, "popped from stack"
   print stack
```

```
Processing a Stack
The list is: 0
The list is: 1 0
The list is: 2 1 0
The list is: 3 2 1 0
3 popped from stack
The list is: 2 1 0
2 popped from stack
The list is: 1 0
1 popped from stack
The list is: 0
0 popped from stack
The list is empty
```

Fig. 22.8 Simple stack implementation—**fig22_08.py**.

22.5 Queues

A *queue* is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced. Queue nodes are removed only from the *head* of the queue and are inserted only at the *tail* of the queue. For this reason, a queue is referred to as a *first-in, first-out (FIFO)* data structure. The insert and remove operations are known as ***enqueue*** and ***dequeue***.

Queues have many applications in computer systems. Most computers have only a single processor, so only one user at a time can be served. Entries for the other users are placed in a queue. Each entry gradually advances to the front of the queue as users receive service. The entry at the front of the queue is the next to receive service.

Queues are also used to support print spooling. A multiuser environment may have only a single printer. Many users may be generating outputs to be printed. If the printer is busy, other outputs may still be generated. These are “spooled” to disk (much as thread is wound onto a spool) where they wait in a queue until the printer becomes available.

Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet’s final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

A file server in a computer network handles file access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.

Figure 22.9 creates class **Queue** primarily through inheritance of class **List** of Fig. 22.3. We want the **Queue** to have methods **enqueue** and **dequeue**. We note that these are essentially the **insertAtBack** and **removeFromFront** methods of class **List**. When we implement the **Queue**’s methods, we have each of these call the appropriate method of class **List**—**enqueue** calls **insertAtBack** and **dequeue** calls **removeFromFront**. Of course, class **List** contains other methods (i.e., **insertAtFront** and **removeFromBack**) that we would not use when manipulating instances of class **Queue**. The main portion of the program uses class **Queue** to instantiate a queue instance. We enqueue integer values 0 through 3, then dequeue the values in first-in, first-out order.

```
# Fig. 22.9: Queue.py
# Class Queue definition

from List import List

class Queue ( List ):
   "Queue built from linked list"

   def enqueue( self, data ):
      "Enqueue element"

      self.insertAtBack( data )

   def dequeue( self ):
```

Fig. 22.9 Simple queue implementation—**Queue.py**.

```
      "Dequeue element"

      return self.removeFromFront()
```

Fig. 22.9 Simple queue implementation—**`Queue.py`**.

```
# Fig. 22.9: fig22_09.py
# Driver to test class Queue

import Queue

queue = Queue.Queue()

print "Processing a Queue"

for i in range( 4 ):
   queue.enqueue( i )
   print queue

while not queue.isEmpty():
   dequeue = queue.dequeue()
   print dequeue, "dequeued"
   print queue
```

```
Processing a Queue
The list is: 0
The list is: 0 1
The list is: 0 1 2
The list is: 0 1 2 3
0 dequeued
The list is: 1 2 3
1 dequeued
The list is: 2 3
2 dequeued
The list is: 3
3 dequeued
The list is empty
```

Fig. 22.9 Simple Queue implementation—**`fig22_09.py`**.

22.6 Trees

Linked lists, stacks and queues are *linear data structures.* A tree is a nonlinear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses *binary trees* (Fig. 22.10)—trees whose nodes all contain two links (one or both of which may be **`None`**). The *root node* is the first node in a tree. Each link in the root node refers to a *child.* The *left child* is the root node of the *left subtree,* and the *right child* is the root node of the *right subtree.* The children of a single node are called *siblings*. A node with no children is called a *leaf node.* Computer scientists normally draw trees from the root node down—exactly the opposite of trees in nature.

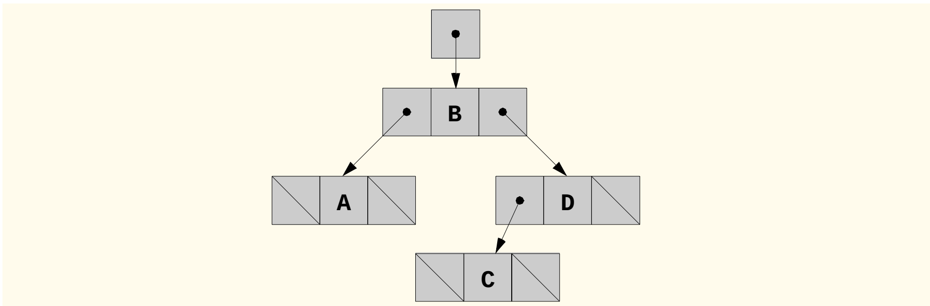


Fig. 22.10 A graphical representation of a binary tree.

In this section, a special binary tree called a *binary search tree (BST)* is created. A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node. Figure 22.11 illustrates a binary search tree with 12 values. Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.



Common Programming Error 22.3

Not setting the links in leaf nodes of a tree to **`None`**.

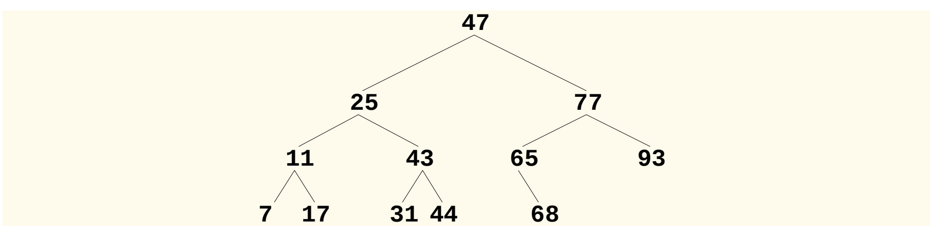


Fig. 22.11 Graphical representation of a binary search tree.

The program of Figure 22.12 creates a binary search tree and traverses it (i.e., walks through all its nodes) three ways—using recursive *inorder*, *preorder* and *postorder traversals*.

```
# Fig. 22.12: Treenode.py
# Treenode definition.

class Treenode:

   def __init__( self, data ):
      "Treenode constructor"

      self.left = None
```

Fig. 22.12 Implementing a binary tree—**`Treenode.py`**.

```
      self.data = data
      self.right = None

   def getData( self ):
      "Get node data"

      return self.data

   def setData( self, newData ):
      "Set node data"

      self.data = newData

   def getLeftNode( self ):
      "Get left child"

      return self.left

   def setLeftNode( self, node ):
      "Set right child"

      self.left = node

   def getRightNode( self ):
      "Get right child"

      return self.right

   def setRightNode( self, node ):
      "Set right child"

      self.right = node
```

Fig. 22.12 Implementing a binary tree—**Treenode.py**.

```
# Fig. 22.12: Tree.py
# Tree definition

from Treenode import Treenode

class Tree:
   "Binary search tree"

   def __init__( self ):
      "Tree Constructor"

      self.rootNode = None

   def insertNode( self, value ):
      "Insert node into tree"

      if self.rootNode is None:            # tree is empty
         self.rootNode = Treenode( value )
```

Fig. 22.12 Implementing a binary tree—**Tree.py**.

```
      else:                              # tree is not empty
         self.insertNodeHelper( self.rootNode, value )

   def insertNodeHelper( self, node, value ):
      "Recursive helper method"

      if value < node.getData():         # insert to left

         if node.getLeftNode() is None:
            node.setLeftNode( Treenode( value ) )
         else:
            self.insertNodeHelper ( node.getLeftNode(), value )

      elif value > node.getData():

         if node.getRightNode() is None:   # insert to right
            node.setRightNode( Treenode( value ) )
         else:
            self.insertNodeHelper ( node.getRightNode(), value )

      else:                              # node duplicate
         print value, "duplicate"

   def preOrderTraversal( self ):
      "Preorder traversal"

      self.preOrderHelper( self.rootNode )

   def preOrderHelper( self, node ):
      "Preorder traversal helper function"

      if node is not None:
         print node.getData(),
         self.preOrderHelper( node.getLeftNode() )
         self.preOrderHelper( node.getRightNode() )

   def inOrderTraversal( self ):
      "Inorder traversal"

      self.inOrderHelper( self.rootNode )

   def inOrderHelper( self, node ):
      "Inorder traversal helper function"

      if node is not None:
         self.inOrderHelper( node.getLeftNode() )
         print node.getData(),
         self.inOrderHelper( node.getRightNode() )

   def postOrderTraversal( self ):
      "Postorder traversal"

      self.postOrderHelper( self.rootNode )

```

Fig. 22.12 Implementing a binary tree—**Tree.py**.

```
    def postOrderHelper( self, node ):
        "Postorder traversal helper function"

        if node is not None:
            self.postOrderHelper( node.getLeftNode() )
            self.postOrderHelper( node.getRightNode() )
            print node.getData(),
```

Fig. 22.12 Implementing a binary tree—**Tree.py**.

```
# Fig. 22.12: fig22_12.py
# The driver to test Tree class.

from Tree import Tree

tree = Tree()
values = raw_input( "Enter 10 integer values:\n" )

for i in values.split():
    tree.insertNode( int( i ) )

print "\nPreorder Traversal"
tree.preOrderTraversal()
print

print "Inorder Traversal"
tree.inOrderTraversal()
print

print "Postorder Traversal"
tree.postOrderTraversal()
print
```

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder Traversal
50 25 12 6 13 33 75 67 68 88
Inorder Traversal
6 12 13 25 33 50 67 68 75 88
Postorder Traversal
6 13 12 33 25 68 67 88 75 50
```

Fig. 22.12 Implementing a binary tree—**fig22_12.py**.

The main program begins by instantiating a binary tree. The program prompts for 10 integers, each of which is inserted in the binary tree through a call to **insertNode**. The program then performs preorder, inorder and postorder traversals (these are explained shortly) of **tree**.

Now we discuss the class definitions. Class **TreeNode** has as data the node's **data** value, and references **left** (to the node's left subtree) and **right** (to the node's right subtree). The constructor sets member **data** to the value supplied as a constructor argument,

and sets references **left** and **right** to **None** (thus initializing this node to be a leaf node). Method **getData** returns the **data** value, and method **setData** sets the data value.

Class **Tree** has data **rootNode**, a reference to the root node of the tree. The class has methods **insertNode** (that inserts a new node in the tree,) and **preorderTraversal**, **inorderTraversal** and **postorderTraversal**, each of which walks the tree in the designated manner. Each of these methods calls its own separate recursive utility method to perform the appropriate operations on the internal representation of the tree. The **Tree** constructor initializes **rootNode** to **None** to indicate that the tree is initially empty.

The **Tree** class' utility method **insertNodeHelper** recursively inserts a node into the tree. *A node can only be inserted as a leaf node in a binary search tree.* If the tree is empty, a new **TreeNode** is created, initialized and inserted in the tree.

If the tree is not empty, the program compares the value to be inserted with the **data** value in the root node. If the insert value is smaller, the program recursively calls **insertNodeHelper** to insert the value in the left subtree. If the insert value is larger, the program recursively calls **insertNodeHelper** to insert the value in the right subtree. If the value to be inserted is identical to the data value in the root node, the program prints the message **"duplicate"** and returns without inserting the duplicate value into the tree.

Each of the methods **inOrderTraversal**, **preOrderTraversal** and **postOrderTraversal** traverse the tree (Fig. 22.13) and print the node values.

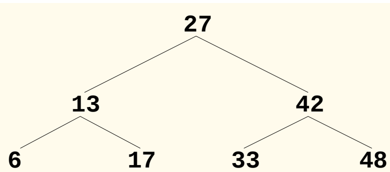


Fig. 22.13 A binary search tree.

The steps for an **inOrderTraversal** are:

1. Traverse the left subtree with an **inOrderTraversal**.
2. Process the value in the node (i.e., print the node value).
3. Traverse the right subtree with an **inOrderTraversal**.

The value in a node is not processed until the values in its left subtree are processed. The **inOrderTraversal** of the tree in Fig. 22.13 is:

```
6 13 17 27 33 42 48
```

Note that the **inOrderTraversal** of a binary search tree prints the node values in ascending order. The process of creating a binary search tree actually sorts the data—and thus this process is called the *binary tree sort*.

The steps for a **preOrderTraversal** are:

1. Process the value in the node.
2. Traverse the left subtree with a **preOrderTraversal**.
3. Traverse the right subtree with a **preOrderTraversal**.

The value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed, and then the values in the right subtree are processed. The **`preOrderTraversal`** of the tree in Fig. 22.13 is:

```
27 13 6 17 42 33 48
```

The steps for a **`postOrderTraversal`** are:

1. Traverse the left subtree with a **`postOrderTraversal`**.
2. Traverse the right subtree with a **`postOrderTraversal`**.
3. Process the value in the node.

The value in each node is not printed until the values of its children are printed. The **`postOrderTraversal`** of the tree in Fig. 22.13 is:

```
6 17 13 33 48 42 27
```

The binary search tree facilitates *duplicate elimination*. As the tree is being created, an attempt to insert a duplicate value will be recognized because a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did. Thus, the duplicate will eventually be compared with a node containing the same value. The duplicate value may be discarded at this point.

Searching a binary tree for a value that matches a key value is fast. If the tree is *balanced*, then each level contains about twice as many elements as the previous level. So a binary search tree with n elements would have a maximum of $\log_2 n$ levels, and thus a maximum of $\log_2 n$ comparisons would have to be made either to find a match or to determine that no match exists. This means, for example, that when searching a (balanced) 1000-element binary search tree, no more than 10 comparisons need to be made because $2^{10} > 1000$. When searching a (balanced) 1,000,000-element binary search tree, no more than 20 comparisons need to be made because $2^{20} > 1{,}000{,}000$.

In the exercises, algorithms are presented for several other binary tree operations such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a level-order traversal of a binary tree. The *level-order traversal* of a binary tree visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, the nodes are visited from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

SUMMARY

- Self-referential classes contain members called links that point to objects of the same class type.
- Self-referential classes enable many objects to be linked together in stacks, queues lists and trees.
- A linked list is a linear collection of self-referential class objects.
- A linked list is a dynamic data structure—the length of the list increases or decreases as necessary.
- Linked lists can continue to grow until memory is exhausted.
- Linked lists provide a mechanism for insertion and deletion of data by reference manipulation.

- A singly linked list begins with a link to the first node, and each node contains a link to the next node "in sequence." This list terminates with a node whose reference member is None. A singly linked list may be traversed in only one direction.
- A circular, singly linked list begins with a link to the first node, and each node contains a link to the next node. The link in the last node references the first node, thus closing the "circle."
- A doubly linked list allows traversals both forwards and backwards. Each node has both a forward link to the next node in the list in the forward direction, and a backward link to the next node in the list in the backward direction.
- In a circular, doubly linked list, the forward link of the last node points to the first node, and the backward link of the first node points to the last node, thus closing the "circle."
- Stacks and queues are constrained versions of linked lists.
- New stack nodes are added to a stack and are removed from a stack only at the top of the stack. For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.
- The link member in the last node of the stack is set to null (zero) to indicate the bottom of the stack.
- The two primary operations used to manipulate a stack are **push** and **pop**. The **push** operation creates a new node and places it on the top of the stack. The **pop** operation removes a node from the top of the stack and returns the popped value.
- In a queue data structure, nodes are removed from the head and added to the tail. For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure. The add and remove operations are known as **enqueue** and **dequeue**.
- Trees are two-dimensional data structures requiring two or more links per node.
- Binary trees contain two links per node.
- The root node is the first node in the tree.
- Each of the references in the root node refers to a child. The left child is the first node in the left subtree, and the right child is the first node in the right subtree. The children of a node are called siblings. Any tree node that does not have any children is called a leaf node.
- A binary search tree has the characteristic that the value in the left child of a node is less than the value in its parent node, and the value in the right child of a node is greater than or equal to the value in its parent node. If there are no duplicate data values, the value in the right child is greater than the value in its parent node.
- An inorder traversal of a binary tree traverses the left subtree inorder, processes the value in the root node and then traverses the right subtree inorder. The value in a node is not processed until the values in its left subtree are processed.
- A preorder traversal processes the value in the root node, traverses the left subtree preorder and then traverses the right subtree preorder. The value in each node is processed as the node is encountered.
- A postorder traversal traverses the left subtree postorder, traverses the right subtree postorder then processes the value in the root node. The value in each node is not processed until the values in both its subtrees are processed.

SUMMARY

[*To be done for second round of review***]**

TERMINOLOGY

[*To be done for second round of review***]**

SELF-REVIEW EXERCISES

[*To be done for second round of review***]**

ANSWERS TO SELF-REVIEW EXERCISES

[*To be done for second round of review***]**

EXERCISES

[*To be done for second round of review***]**

Notes to Reviewers:

- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **`ben.wiedermann@deitel.net`**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copy edited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are mostly concerned with technical correctness and correct use of idiom. We will not make significant adjustments to our writing or coding style on a global scale. Please send us a short e-mail if you would like to make a suggestion.
- If you find something incorrect, please show us how to correct it.
- In the later round(s) of review, please read all the back matter, including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

Additional Comments:

- The goal of this chapter is to teach the concept of data structures. However, it would be a good idea to include more performance tips, throughout, to demonstrate why Python may or may not be the best language in which to actually implement these data structures.
- Currently, we are reorganizing our object-oriented chapters to better capture the Python OOP idiom (specifically, attribute access). The implications for this chapter are:

  1. "Private data"

  1. Access methods go away.

  2. We may be able to prevent access to un-needed base class methods from clients of a derived class?

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

W

[*** Notes to Reviewers ***]

REVIEWERS: This chapter still needs its treatment of wireless device programming. We are working on the section, but have chosen to send out this chapter without that segment and without the back matter—the summary terminology and exercises. When we implement the wireless section and your comments from this first round of review, we are going to send this chapter in its entirety out for a second round of review.

Please be sure to do each of the following items:

1. Read the entire chapter.
2. Please mark your comments in place on a paper copy of the chapter.
3. Please return only marked pages to Deitel & Associates, Inc.
4. The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
5. Run each example.
6. Comment on our example selection and implementation.
7. Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
8. Watch for proper use of idioms. If there are improper uses, state explicitly how to correct them.
9. Suggest other examples or features that should be covered (if necessary).
10. Do we need additional line art or tables. If so, where are they needed and for what are they needed?
11. Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
12. Please be constructive. This book will be published soon. We all want to published the best possible book.
13. If you find something that is incorrect, please show us how to correct it.
14. 13. Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

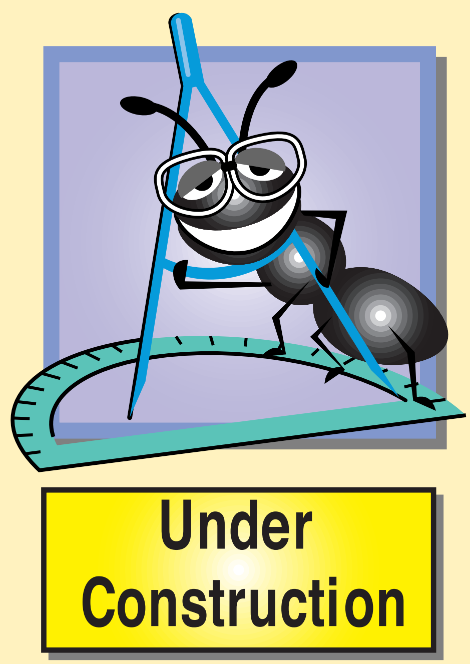
23

Case Study: Online Bookstore

Objectives

- To build a three-tier, client/server, distributed Web application using Python and CGI.
- To understand the concept of an HTTP session.
- To be able to use a **`Session`** class to keep track of an HTTP session between pages.
- To be able to create XML from a script and XSL transformations to convert the XML into a format the client can display.
- To be able to deploy an application on an Apache Web server.

*[*** NEED QUOTES. ***]*



Outline

23.1 Introduction

In this chapter, we implement a bookstore Web application that integrates many technologies we cover in this book while serving as a capstone for our presentation of Python CGI. The technologies used in the application include CGI (Chapter 6), XML, XSL and XSLT (Chapters 15–16), mySQL and the Python DB-API (Chapter 17), HTML and XHTML (Chapters 26–27) and Cascading Style Sheets (Chapter 28). The case study also introduces additional features—we will discuss the new elements as we encounter them. We demonstrate how to deploy this application on an Apache server so that after reading this chapter, you will be able to implement a substantial distributed Web application containing many components on an Apache server.

23.2 HTTP Sessions and Session Tracking Technologies

Web sites that can provide custom Web pages and functionality tailored to clients viewing the content can implement e-commerce applications. One example of such a Web site application is the online shopping cart we are building for this chapter's online bookstore case study. To enable this type of application, the server must distinguish between clients so the company can ship the ordered items and properly charge each client. *Session-tracking* technologies allow servers to distinguish between clients. In this section, we introduce and explain cookies and session ID technologies and how they operate using Internet protocols.

The Internet uses the *HyperText Transfer Protocol (HTTP)*, a *connectionless protocol*. A connectionless protocol is one in which every request made from a Web browser to a server uses a new connection, and once a client request is processed, the connection termi-

nates. This means that the client must identify itself with each request while connecting to the server using HTTP.

One session tracking method uses *cookies*. Cookies are small text files sent by a Python CGI script as part of a response to a client. Cookies can store information on the client's computer for retrieval later in the same browsing session or in future browsing sessions. For example, because cookies can be retrieved later in the same session, cookies could be used in a shopping application to indicate the client's preferences because the cookies have traced the clients' movements—which pages have been visited and what links have been clicked. When the Python script receives the client's next communication, the Python script can examine the cookie(s) information and identify the client's preferences and display products that may be of interest to the client, based the pages they have viewed.

Every HTTP-based interaction between a client and a server includes a *header* that contains information about the *request* (communication from the client to the server) or information about the *response* (communication from the server to the client). When a Python script receives a request, the header includes information such as the request type (e.g., **`GET`** or **`POST`**) and cookies stored on the client machines by the server. When the server formulates its response, the header information includes any cookies the server will store on the client computer.

Depending on the *maximum age* of a cookie, the Web browser either maintains the cookie for the duration of the browsing session (i.e., until the user closes the Web browser) or stores the cookie on the client computer to access in a future session. When the browser makes a request of a server, cookies previously sent to the client by that server are returned to the server (if they have not expired) as part of the request formulated by the browser. Cookies are automatically deleted when they *expire* (i.e., reach their maximum age).

Cookies often are the easiest way for a Python programmer to distinguish clients. However, cookies are not accepted by all client types or browsers. Also, users may disable cookies, which may make users unable to view content on cookie-dependent sites—some sites require cookies for clients to even access home pages. For these reasons, we have chosen not to use cookies to track sessions in our online bookstore.

Portability Tip 23.1

Not all browsers support cookies. Designing a server which uses cookies may exclude some users from accessing your site.

Another method for session tracking involves embedding state information. The first time a client connects to a server, it is assigned a unique *session ID* by the server. When the client makes additional requests, the client's session ID is compared against the session IDs stored on the server.

The ID must be passed from page to page so each Web page file will know the session ID of the current client, thereby distinguishing clients. This can be done in different ways. One method of passing the ID is to place a hidden form field. Then the next page can access the ID as a normal CGI parameter. Another method is to add the ID to the URL by adding the ID to a hyperlink that points to the next page. The next page can then extract the ID from the URL. If the ID is appended to the URL as part of a query string, however, the next page can access the ID as a normal CGI parameter.

Although more extensible than cookies, tracking session information using embedded session IDs has disadvantages. One disadvantage to this method is that it creates Web page

addresses much longer than they normally would be when the session ID is embedded in every hyperlink. Embedding information also presents a potential security risk. Storing the session ID in the web page or URL creates the possibility that a person other than the user may see the ID and gain access to the user's data. Nonetheless, we have chosen this method to track HTTP sessions in our online bookstore.



Good Programming Practice 23.1

Every session-tracking method has advantages and disadvantages. Research and carefully consider each technique before selecting one for a site.

23.3 Tracking Sessions with Python `Session` Class

Before we begin executing scripts, we are going to introduce the class we use to track sessions in our bookstore application. In this section, we will explain our use of the **`Session`** class defined in **`Session.py`** to track an HTTP session (Fig. 23.1). We discuss how a script can specify whether to create a new session when a script creates a **`Session`** object. If the script creates a new session, a new session ID is created and a new dictionary of session data is initialized. Otherwise, **`Session`** extracts the session ID from the query string and loads the session data for that ID. Session data is pickled and stored on the server. You will see class **`Session`** executed in Figure 23.6.

```
# Fig. 23.1: Session.py
# Contains a Session class that keeps track of an http session
# by assigning a session ID and pickling session information.

import os
import re
import md5
import cgi
import time
import urlparse
import os.path
import cPickle
from UserDict import UserDict

def getClientType():
   """Return the client type and file extension"""

   if re.search( "MSIE", os.environ[ "HTTP_USER_AGENT" ] ):
      return ( "html", "html" )
   elif re.search( "Netscape", os.environ[ "HTTP_USER_AGENT" ] ):
      return ( "html", "html" )
   elif re.search( "text/vnd.wap.wml",
      os.environ[ "HTTP_ACCEPT" ] ):
      return ( "wml", "wml" )
   else:
      return ( "html_basic", "html" )

def getContentType():
   """Return the contents of the client's contentType file"""
```

Fig. 23.1 Utility functions and **`Session`** class that track an http session.

```python

   try:
      file = open( getClientType()[ 0 ] + "/contentType.txt" )
   except:
      raise SessionError( "Missing+content+type+file" )

   contentType = file.read()
   file.close()
   return contentType

def redirect( URL ):
   """Redirect the client to a relative URL"""

   print "Location: %s\n" % \
      urlparse.urljoin( "http://" + os.environ[ "HTTP_HOST" ] +
         os.environ[ "REQUEST_URI" ], URL )

class SessionError( Exception ):
   """User-defined exception for Session class"""

   def __init__( self, error ):
      """Set error message"""

      self.error = error

   def __str__( self ):
      """Return error message"""

      return self.error

class Session( UserDict ):
   """Session class keeps tracks of an HTTP session"""

   def __init__( self, createNew = 0 ):
      """Create a new session or load an existing session"""

      # attempt to load previously created session
      if not createNew:

         # session ID is passed in query string
         queryString = cgi.parse_qs( os.environ[ "QUERY_STRING" ] )

         # no ID has been supplied in query string
         if not queryString.has_key( "ID" ):
            raise SessionError( "No+ID+given" )

         self.sessionID = queryString[ "ID" ][ 0 ]
         self.fileName = os.getcwd() + "/sessions/." + \
            self.sessionID

         # supplied ID is invalid
         if not self.sessionExists():
            raise SessionError( "Nonexistant+ID+given" )

```

Fig. 23.1 Utility functions and **Session** class that track an http session.

```
         # load pickled session dictionary
         UserDict.__init__( self, self.loadSession() )

      # create new session
      else:
         self.sessionID = self.generateID()
         self.fileName = os.getcwd() + "/sessions/." + \
            self.sessionID

         if self.sessionExists():
            raise SessionError( "Session+already+exists" )

         UserDict.__init__( self )   # dictionary is empty

         # add ID, agent type, content type and empty cary to data
         self.data[ "ID" ] = self.sessionID
         self.data[ "agent" ], self.data[ "extension" ] = \
            getClientType()
         self.data[ "content type" ] = getContentType()
         self.data[ "cart" ] = {}

   def sessionExists( self ):
      """Determine if the specified session file exists"""

      return os.path.exists( self.fileName )

   def loadSession( self ):
      """Return unpickled dictionary of existing session"""

      if self.sessionExists():
         sessionFile = open( self.fileName )
         data = cPickle.load( sessionFile )
         sessionFile.close()
         return data

   def saveSession( self ):
      """Pickle session dictionary to session file"""

      sessionFile = open( self.fileName, "w" )
      cPickle.dump( self.data, sessionFile )
      sessionFile.close()

   def deleteSession( self ):
      """Delete session file"""

      os.remove( self.fileName )

   def generateID( self ):
      """Use md5 to generate a unique ID"""

      seed = str( time.time() ) + os.environ[ "REMOTE_ADDR" ] + \
         os.environ[ "REMOTE_PORT" ]
      ID = md5.new( seed )
```

Fig. 23.1 Utility functions and **Session** class that track an http session.

```
137         return ID.hexdigest()
```

Fig. 23.1 Utility functions and **Session** class that track an http session.

When a **Session** object is created, **createNew** (the argument passed to the contstructor) can be specified to a value other than 0 (the default) to create a new session. In this case, execution begins at line 79 with a call to method **generateID**. Method **generateID** (lines 131–136) uses module **md5** to generate a unique ID. Lines 134–135 create a string from the time of the session, the client address and the client port. Lines 136–137 then create and return a unique ID using this string. For more information on **md5**, review Chapter 21.

When the **Session** obtains its new ID from **generateID**, it stores the name of its session file, **fileName**, and checks if the session already exists. Note that the filename of a session is a period (**.**) followed by the session ID. All session files are stored in a subdirectory, **sessions**, of the current working directory. If the session file already exists, **Session** raises the user-defined exception **SessionError** (line 82).

Class **Session** inherits from class **UserDict**. **UserDict** is a class defined in module **UserDict** that simulates a dictionary. The contents of each instance are stored in a Python dictionary called **data**. Line 85 initializes an instance of **UserDict**, creating an empty session dictionary (**data**). **Data** then stores the session ID (line 89). Lines 100–101 obtain the client type from function **getClientType** and store it in the session dictionary. Function **getClientType** searches the **HTTP_USER_AGENT** environment variable for certain keywords to determine the client type (lines 15–26). Line 91 stores the results of function **getContentType** in **data**. Function **getContentType** opens the **contentType.txt** file, which resides in a subdirectory named after the client type, and returns the contents of the file (lines 25–35). Figure 23.2 contains an example of such a file. Line 96 creates an empty shopping cart (an empty dictionary).

```
Content-type: text/html

```

Fig. 23.2 contentType.txt for html clients.

To save session data between pages, method **saveSession** must be called (lines 119–124). This method creates a new session file corresponding to the value of attribute **fileName**. Line 123 uses module **cPickle** to pickle the session dictionary and dump it into the session file.

To open an existing session from a different script, create a **Session** with **createNew** set to 0 (default). If **createNew** is 0, execution begins in line 67. **Session** obtains the query string and parses it. If no ID is specified, the constructor raises a **SessionError**. Otherwise, the session ID is extracted and the filename is determined (lines 76–78). If the session does not exist, the constructor raises a **SessionError** (lines 81–82). Otherwise, the constructor calls the **UserDict** base-class constructor (line 85). The value of the session dictionary (**data**) is the value returned from method **loadSession**. This method (lines 110–117) opens the session file (line 114). It then uses **cPickle** to unpickle and return the session dictionary it contains (lines 115–117).

When a session is no longer needed, it can be removed from the server by invoking method **deleteSession** (line 126–129). This method deletes the session file by calling **os.remove**.

23.4 Bookstore Architecture

This section overviews the architecture of the bookstore application. We present a diagram of the basic interactions between Python scripts. and a table of the files used in the case study.

The shopping cart case study consists of a series of Python scripts that interact to simulate an online bookstore selling Deitel publications. This case study is implemented as a distributed, three-tier, Web-based application. The client tier is represented by the user's Web browser. The browser displays either static or dynamically created documents that allow the user to interact with the server tier. These documents are created based upon the user's client type. The server tier consists of several scripts that act on behalf of the client. These scripts perform tasks such as creating a list of publications, creating documents containing the details about a publication, adding items to the shopping cart, viewing the shopping cart and processing the final order.

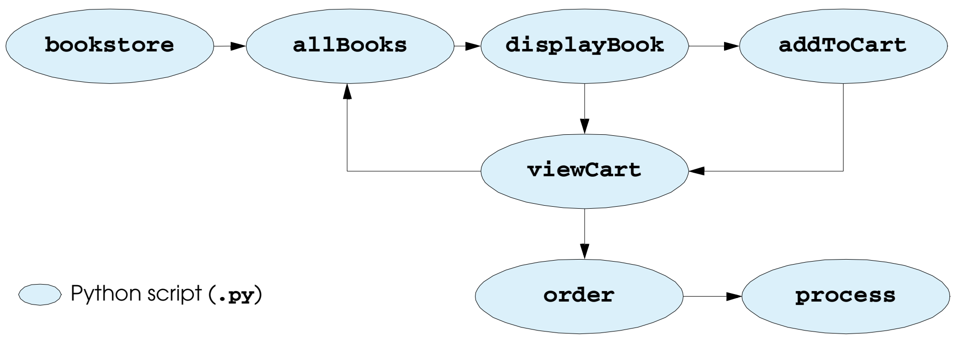


Fig. 23.3 **Bug2Bug.com** bookstore component interactions.

Figure 23.3 illustrates the interactions between the bookstore's application components. After creating a **Session** for the user, the user will be forwarded to **allBooks.py** a script that interacts with a database to create the list of books dynamically. The database tier uses the **books** database. The result is an XML document that represents the list of books. This XML document is then processed against a client-specific XSLT stylesheet to produce a page containing links to **displayBook**. This script receives as a parameter the ISBN number of the selected book and uses the ISBN to retrieve the book data and produce XML that represents the selected book. This XML is then processed against a different client-specific XSLT stylesheet to produce a document containing the information for that book. From this document, the user can use GUI components (in this case, buttons) to place the current book in the shopping cart or view the shopping cart.

Adding a book to a shopping cart invokes **addToCart**. Viewing the cart contents invokes **viewCart** which returns a client-specific document (again, created by processing

XML with XSLT) containing the cart contents, the subtotal dollar cost of each item and the total dollar cost of all the items in the cart. When the user adds an item to the shopping cart, the **addToCart** script processes the user's request, then forwards the request to **viewCart** to create the document that displays the current cart. At this point, the user can either continue shopping (**allBooks.py**) or proceed to checkout (**order.py**). In the latter case, the user is presented with a form to input name, address and credit-card information. Then, the user submits the form to invoke **process.py**, which completes the transaction by sending a confirmation document to the user. Figure 23.4 overviews the scripts and other files used in this case study.

File	Description
Session.py	Contains the **Session** class. An instance of this class keeps track of an HTTP session by assigning each user a unique session ID and pickling a dictionary of data for each ID. It also contains three utility functions for redirecting the client, determining the user's client type and determining the client's content type (stored in **contentType.txt**).
contentType.txt	Contains the line that specifies to the browser the content type of the data. There is one of these files for each client type.
bookstore.py	This is the default home page for the bookstore, which is displayed by entering the following URL in the client's Web browser: **http://localhost/cgi-bin/bookstore/bookstore.py** Here, a new **Session** is created for the user to track the HTTP session. The user is then forwarded to **allBooks.py**.
styles.css	This Cascading Style Sheet (CSS) file is linked to all XHTML and XHTML Basic documents rendered on the client. The CSS file allows us to apply uniform formatting across all the static and dynamic documents rendered.
allBooks.py	This script uses **Book** objects to create a document containing the product list. It queries the **catalog** database to obtain the list of titles in the database. The results are processed and placed into a list of **Book** objects. The list is stored as a session attribute for the client. The script creates an XML document which represents all the books, then applies a client-specific XSLT transformation (**allBooks.xsl**) to the XML to produce a document that can be rendered by the client.
allBooks.xsl	This XSLT style sheet transforms the XML representation of the entire catalog of books into a document that the client browser can render. There is one of these files for each client type.

Fig. 23.4 Components for bookstore case study (part 1 of 3).

File	Description
Book.py	Contains the **Book** class. An instance of this class represents the data for one book. The **Book**'s **getXML** method returns an XML **Element** that represents the book.
displayBook.py	This script obtains the XML representation of a book selected by the user, then applies a client-specific XSLT transformation (**displayBook.xsl**) to the XML to produce a document that can be rendered by diverse clients.
displayBook.xsl	This XSLT style sheet transforms the XML representation of a book into a document that the client browser can render. There is one of these files for each client type.
CartItem.py	Contains the **CartItem** class. An instance of this class maintains a **Book** and the current quantity for that book in the shopping cart. **CartItem**s are stored in a dictionary that represents the shopping cart contents.
addToCart.py	This script updates the shopping cart. If a **CartItem** for the item is already in the cart, the script updates the quantity of that item in the class. Otherwise, the script creates a new **CartItem** with a quantity of 1. After updating the cart, the user is forwarded to **viewCart.py** to view the current cart contents.
viewCart.py	This script extracts the **CartItem**s from the shopping cart, subtotals each item in the cart, totals all the items in the cart and creates an XML document that represents all items in the cart. The script then applies a client-specific XSLT transformation (**viewCart.xsl**) to the XML to produce a document that can be rendered by the client. This process allows the client to view the cart in tabular form.
viewCart.xsl	This XSLT style sheet transforms the XML representation of all of the **CartItem**s in the cart into a document that the client browser can render. There is one of these files for each client type.
order.py	When viewing the cart, the user can click a **Check Out** button to execute this script. This script displays a client-specific order form. In this example, the form has no functionality. However, it is provided to help complete the application.
orderForm.html, **orderForm.wml**	This static document contains an order form. It is displayed by **order.py**.
process.py	This final script pretends to process the user's credit-card information and loads a client-specific document indicating that the order was processed and the total order value.
thankYou.html, **thankYou.wml**	This static document, displayed by **process.py**, contains a message that the order was processed and the total order value.

Fig. 23.4 Components for bookstore case study (part 2 of 3).

File	Description
`error.py`	This script executes when an error occurs. It creates an XML document which represents the error. It then processes the XML against a client-specific XSLT style sheet (**`error.xsl`**) to produce a document that can be rendered by the client. This document indicates to the user the error that occurred.
`error.xsl`	This XSLT style sheet transforms the XML representation of all of an error into a document that the client browser can render. There is one of these files for each client type.

Fig. 23.4 Components for bookstore case study (part 3 of 3).

23.5 Setting up the Bookstore

All bookstore files are located in the Chapter 23 folder of the CD that accompanies this book. To set up the bookstore on your Web server, first copy subfolder **`bookstore`** and its contents into your server's root directory (i.e., **`htdocs`** for Apache). Next, copy the contents of subfolder **`cgi-bin`** (subfolder **`bookstore`**) into your Web server's cgi-bin. Finally, restart your Web server. Figure 23.5 illustrates the directory structure for Apache.

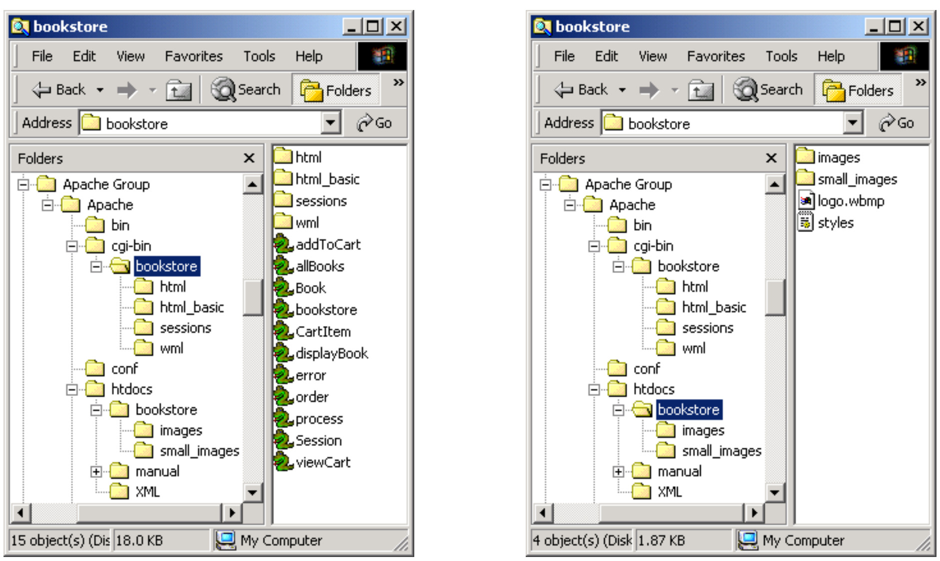


Fig. 23.5 Apache directory structure

23.6 Entering the Bookstore

Figure 23.6 (**bookstore.py**) is the default home page for the bookstore. It is the only way to enter the bookstore. When the site is running on Apache on your computer, enter the following URL in your Web browser to enter the bookstore:

```
http://localhost/cgi-bin/bookstore/bookstore.py
```

```
1  #!c:\Python\python.exe
2  # Fig. 23.5: bookstore.py
3  # Create a new Session for client.
4
5  import sys
6  import time
7  import Session
8
9  # create new Session
10 try:
11    session = Session.Session( 1 )
12 except Session.SessionError, message:   # ID already exists
13    time.sleep( 1 )                      # wait 1 second
14    Session.redirect( "bookStore.py" )   # try again
15    sys.exit()
16
17 # re-direct to allBooks.py
18 nextPage = "allBooks.py?ID=%s" % session[ "ID" ]
19 session.saveSession()
20 Session.redirect( nextPage )
```

Fig. 23.6 Bookstore home page (**bookstore.py**).

Line 11 creates a new **Session** for the client (see Section 11.2). Recall that if the session-generated ID already exists, a **SessionError** is raised. In this case, the program sleeps for one second (so that the seed for **md5** changes), redirects the client to **bookstore.py** (to make another attempt) and exits (lines 13–15). Otherwise, lines 18–20 are executed. Line 18 creates the redirection string to send the client to **allBooks.py**. Note that the session ID is stored in the URL as part of the query string. This ensures **allBooks.py** can determine the client's identity. Lines 19–20 save the session and print the redirection string, sending the client to **allBooks.py**.

23.7 Obtaining the Book List from the Database

We must first create a representation for a single book before we can obtain a list of books (Fig. 23.7). An instance of the **Book** class represents the properties for one book, including the book's ISBN, title, copyright, cover image file name, edition number, publisher ID number and price although some of this information is not used in this example. Each property is a read/write property. **Book** method **getXML** returns an XML **Element** representing the book.

```python
# Fig. 23.6: Book.py
# Represents one book.

class Book:
   """A Book object contains the data for one book"""

   def setISBN( self, isbn ):
      """Set ISBN number"""

      self.ISBN = isbn

   def getISBN( self ):
      """Return IBSN number"""

      return self.ISBN

   def setTitle( self, bookTitle ):
      """Set book title"""

      self.title = bookTitle

   def getTitle( self ):
      """Return book title"""

      return self.title

   def setCopyright( self, year ):
      """Set copyright year"""

      self.copyright = year

   def getCopyright( self ):
      """Return copyright year"""

      return self.copyright

   def setImageFile( self, filename ):
      """Set file name of image representing product cover"""

      self.imageFile = filename

   def getImageFile( self ):
      """Return file name of image representing product cover"""

      return self.imageFile

   def setEditionNumber( self, edition ):
      """Set edition number"""

      self.editionNumber = edition

```

Fig. 23.7 **Book** that represents a single book's information and defines the XML format of that information (part 1 of 3).

```
   def getEditionNumber( self ):
      """Return edition number"""

      return self.editionNumber

   def setPublisherID( self, id ):
      """Set publisher ID number"""

      self.publisherID = id

   def getPublisherID( self ):
      """Return publisher ID number"""

      return self.publisherID

   def setPrice( self, amount ):
      """Set price"""

      self.price = amount

   def getPrice( self ):
      """Return price"""

      return self.price

   def getXML( self, document ):
      """Return an XML representation of the product"""

      # create product node
      product = document.createElement( "product" )

      # create isbn element, append as child of product
      temp = document.createElement( "isbn" )
      temp.appendChild(
         document.createTextNode( self.getISBN() ) )
      product.appendChild( temp )

      # create title element, append as child of product
      temp = document.createElement( "title" )
      temp.appendChild( document.createTextNode(
         self.getTitle() ) )
      product.appendChild( temp )

      # create price element, append as child of product
      temp = document.createElement( "price" )
      temp.appendChild( document.createTextNode(
         self.getPrice() ) )
      product.appendChild( temp )

      # create imageFile element, append as child of product
      temp = document.createElement( "imageFile" )
      temp.appendChild( document.createTextNode(
         self.getImageFile() ) )
```

Fig. 23.7 **Book** that represents a single book's information and defines the XML format of that information (part 2 of 3).

```
105         product.appendChild( temp )
106
107         # create copyright element, append as child of product
108         temp = document.createElement( "copyright" )
109         temp.appendChild( document.createTextNode(
110            self.getCopyright() ) )
111         product.appendChild( temp )
112
113         # create publisherID element, append as child of product
114         temp = document.createElement( "publisherID" )
115         temp.appendChild( document.createTextNode(
116            self.getPublisherID() ) )
117         product.appendChild( temp )
118
119         # create editionNumber element, append as child of product
120         temp = document.createElement( "editionNumber" )
121         temp.appendChild( document.createTextNode(
122            self.getEditionNumber() ) )
123         product.appendChild( temp )
124
125         return product
```

Fig. 23.7 **Book** that represents a single book's information and defines the XML format of that information (part 3 of 3).

Method **getXML** (lines 77–124) uses the DOM **Document** and **Element** interfaces to create an XML representation of the book data as part of the **Document** that is passed as an argument to the method. The complete information for one book is placed in a **product** element (created in line 81). The elements for the individual properties of a book are appended to the **product** element as children. For example, line 84 uses **Document** method **createElement** to create element **isbn**. Line 85 uses **Document** method **createTextNode** to specify the text in the **isbn** element, and uses **Element** method **appendChild** to append the text to element **isbn**. Then, line 86 appends element **isbn** as a child of element **product** with **Element** method **appendChild**. Similar operations are performed for the other book properties. Line 124 returns element **product** to the caller. For more information about XML and Python, refer to Chapters 15 and 16.

Recall that after creating a session for the client, **bookstore.py** redirects the user to **allBooks.py**. This program retrieves the list of books from the **catalog** database and dynamically generates an XML document that represents it. This document is then processed against a client-specific XSLT stylesheet called **allBooks.xsl**. The results are then rendered on the client.

```
#!c:\Python\python.exe
# Fig. 23.7: allBooks.py
# Retrieve all books from database and store in session.
# Display book list to client by retrieving XML and converting
# to required format using client-specific XSLT stylesheet.

import sys
```

Fig. 23.8 **allBooks.py** returns to the client a document containing the book list (part 1 of 3).

```python
import Book
import Session
import MySQLdb
from xml.xslt import Processor
from xml.dom.DOMImplementation import implementation

# load Session
try:
   session = Session.Session()
except Session.SessionError, message:   # invalid/no session ID
   Session.redirect( "error.py?message=%s" % message )
   sys.exit()

# setup mySQL statement
query = """SELECT isbn, title, editionNumber,
   copyRight, publisherID, imageFile, price
   FROM titles ORDER BY title"""

# attempt database connection and retrieve list of Books
try:

   # connect to the database, retrieve a cursor and execute query
   connection = MySQLdb.connect( db = "books" )
   cursor = connection.cursor()
   cursor.execute( query )

   # acquire results and close database connection
   results = cursor.fetchall()
   cursor.close()
   connection.close()
except OperationalError, message:
   Session.redirect( "error.py?message=%s" % message )
   sys.exit()

allBooks = []

# Get row data
for row in results:
   book = Book.Book()
   book.setISBN( row[ 0 ] )
   book.setTitle( row[ 1 ] )
   book.setEditionNumber( str( row[ 2 ] ) )
   book.setCopyright( row[ 3 ] )
   book.setPublisherID( str( row[ 4 ] ) )
   book.setImageFile( row[ 5 ] )
   book.setPrice( str( row[ 6 ] ) )

   allBooks.append( book )

session[ "titles" ] = allBooks

# genereate XML
document = implementation.createDocument( None, None, None )
```

Fig. 23.8 **allBooks.py** returns to the client a document containing the book list (part 2 of 3).

```
61   catalog = document.createElement( "catalog" )
62   document.appendChild( catalog )
63
64   # add all products to catalog
65   for book in allBooks:
66      catalog.appendChild( book.getXML( document ) )
67
68   # process XML against XSLT stylesheet
69   processor = Processor.Processor()
70   style = open( session[ "agent" ] + "/allBooks.xsl" )
71   processor.appendStylesheetString( style.read() % session[ "ID" ] )
72   results = processor.runNode( document )
73   style.close()
74
75   # display content type and processed XML
76   pageData = session[ "content type" ] + results
77   session.saveSession()    # save Session data
78   print pageData
```

Fig. 23.8 **allBooks.py** returns to the client a document containing the book list (part 3 of 3).

Lines 15–19 load the session. If the session ID is not specified in the query string or if the specified ID is invalid, the user is redirected to the error message that displays **in error.py**.

Lines 22–24 prepare the mySQL statement that **allBooks** uses to query the **catalog** database. Lines 30–37 then connect to the database and retrieve the list of books. If an error occurs, the user is redirected to **error.py** and the program exits (lines 39–40).

Lines 45–55 create a **Book** object is created for each book in the database, its attributes are set and appended to list **allBooks** (lines 45–55). Note that the edition number, publisher ID, and price attributes must first be converted to strings. This is because the values are stored as integer and float values in the database; however, each **Book**'s **getXML** method creates a **TextNode** for each of these attributes and **createTextNode** only accepts strings. Line 57 stores the list of **Book** objects in the session dictionary with key **titles**.

We then create an XML **Document** representing the entire catalog of books. Line 60 uses the **createDocument** method of **xml.dom.DOMImplementation.implementation** to create a blank DOM **Document** called **document**. **Document** method **createElement** creates the **catalog** element (line 61). Line 62 appends the **catalog** element to **document**. Lines 65–66 retrieve the **product** element for each book and use method **appendChild** to append the element to **catalog**.

A client-specific XSLT stylesheet processes the XML **Document** (lines 69–73). An XSLT **Processor** is created (line 69) and retrieves the XSLT stylesheet called **allBooks.xsl** (line 70). Note that the copy of **allBooks.xsl** opened is the one found in the directory named after the client type. This ensures that the XSLT stylesheet will transform our XML **Document** into a format that is accepted by various clients. Line 71 appends the stylesheet to the list of stylesheets the processor may use. The session ID must be inserted into the stylesheet first, because the ID is not contained in the XML **Document** that the stylesheet will transform. Lines 72–73 run the processor on **document** and close the stylesheet file, respectively. We then display the transformed XML to the client. Line

76 creates the string that contains the content type specification and the processor results. Lines 77 and 78 save the session and display the page to the user.

Figure 23.9 contains the XSLT stylesheet used to transform the XML catalog representation into XHTML. The resulting XHTML document is shown in the screen capture in of Fig. 23.9.

```
<?xml version = "1.0"?>
<!-- Fig. 23.8: allBooks.xsl                             -->
<!-- XSLT stylesheet that transforms XML generated by   -->
<!-- books.py into XHTML.                               -->
<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <xsl:output method = "xml" omit-xml-declaration = "no"
      indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
      doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>

   <!-- template for catalog element -->
   <xsl:template match = "catalog">
      <html xmlns = "http://www.w3.org/1999/xhtml"
            xml:lang = "en" lang = "en">

      <head>
         <title>Book List</title>
         <link rel = "stylesheet" href = "/bookstore/styles.css"
            type = "text/css" />
      </head>

      <body>

         <p class = "bigFont">Available Books</p>
        <p class = "bold">Click a link to view book information</p>

         <!-- match product elements to product template -->
         <xsl:apply-templates select = "/catalog/product">

            <!-- sort products by title -->
            <xsl:sort select = "title"/>

         </xsl:apply-templates>

      </body>
      </html>
   </xsl:template>

   <!-- template for building row of Product information -->

   <xsl:template match = "product">

      <a href = "displayBook.py?ID=%s&amp;isbn={isbn}">
         <strong><xsl:value-of select = "title"/>, <xsl:value-of
```

Fig. 23.9 **allBooks.xsl** for an HTML client type which transforms the XML representation of the catalog into XHTML.

```
46            select = "editionNumber"/>e</strong>
47         </a><br/>
48
49      </xsl:template>
50   </xsl:stylesheet>
```



Fig. 23.9 **allBooks.xsl** for an HTML client type which transforms the XML representation of the catalog into XHTML.

An **xsl:template** defines **catalog** elements (lines 13–38). Within this template, we insert any matches of the **product** template (lines 29–34). These matches are sorted by their **title** element (line 32). This ensures the book list appears in alphabetical order by the title name when it generates.

Lines 42–49 define an **xsl:template** for elements named **product**. Line 44 specifies a **anchor** tag with attribute **href**. The value of the **href** attribute is specified to be a reference to **displayBook.py** with a query string containing the session ID and the value of the **isbn** element of an XML document (accessed by **{isbn}**). This ensures that **displayBook** will be able to identify the client as well as the book to display. The anchor tag contains text and the values of the **title** and **edition** elements of an XML document (lines 45–46

The XSL document specifies a linked style sheet **styles.css** (Fig. 23.10). All XHTML documents sent to the client use this style sheet, so that uniform formatting can be applied to the documents. Lines 1–2 indicate that all text in the **body** element should be centered and that the background color of the body should be steel blue. The background

color is represented by the hexadecimal number **#b0c4de**. Line 3 defines class **.bold** to apply bold font weight to text. Lines 4–7 define class **.bigFont** with four CSS attributes. Elements to which this class is applied appear in the bold, Helvetica font which is double the size of the base-text font. The color of the font is dark blue (represented by the hexadecimal number **#00008b**). If Helvetica font is not available, the browser will attempt to use Arial, then the generic font **sans-serif** as a last resort. Class **.italic** applies italic font style to text (line 8). Class **.right** right justifies text (line 9). Lines 10–11 indicate that all **table**, **th** (table head data) and **td** (table data) elements should have a three-pixel, grooved border with five pixels of internal padding between the text in a table cell and the border of that cell. Lines 12–14 indicate that all **table** elements should have bright blue background color (represented by the hexadecimal number **#6495ed**), and that all **table** elements should use automatically determined margins on both their left and right sides. This causes the table to be centered on the page. Not all of these styles are used in every XHTML document. However, using a single linked style sheet allows us to change the look and feel of our store quickly and easily by modifying the CSS file. For more information on CSS see Chapter 28.

Portability Tip 23.2

Different browsers have different levels of support for Cascading Style Sheets.

```
1   body            { text-align: center;
2                     background-color: #boc4de }
3   .bold           { font-weight: bold }
4   .bigFont        { font-family: helvetica, arial, sans-serif;
5                     font-weight: bold;
6                     font-size: 2em;
7                     color: #00008b }
8   .italic         { font-style: italic }
9   .right          { text-align: right }
10  table, th, td   { border: 3px groove;
11                    padding: 5px }
12  table           { background-color: #6495ed;
13                    margin-left: auto;
14                    margin-right: auto }
```

Fig. 23.10 Shared cascading style sheet (**styles.css**) used to apply common formatting across XHTML documents rendered on the client.

23.8 Viewing a Book's Details

Selecting a book in **allBooks.py** forwards the user to **displayBook.py**. This program extracts the ISBN from the query string determines what book the user has selected by. It then obtains the XML representation of the book and processes it against a client-specific XSLT stylsheet (**displayBook.xsl**). The results are sent to the user.

```
#!c:\Python\python.exe
```

Fig. 23.11 **displayBook.py** converts the XML representation of the selected book to a client-specific format using an XSLT stylesheet (part 1 of 3).

```
# Fig. 23.10: displayBook.py
# Retrieve one book's XML representation, convert
# to required format using client-specific XSLT
# stylesheet and display results.

import cgi
import sys
import Session
from xml.xslt import Processor
from xml.dom.DOMImplementation import implementation

form = cgi.FieldStorage()

# ISBN has not been specified
if not form.has_key( "isbn" ):
   Session.redirect( "error.py?message=No+ISBN+given" )
   sys.exit()

# load Session
try:
   session = Session.Session()
except Session.SessionError, message:   # invalid/no session ID
   Session.redirect( "error.py?message=%s" % message )
   sys.exit()

titles = session[ "titles" ]     # get titles
session[ "bookToAdd" ] = None    # book has not been found

# locate Book object for selected book
for book in titles:

   if form[ "isbn" ].value == book.getISBN():
      session[ "bookToAdd" ] = book
      break

# book has been found
if session[ "bookToAdd" ] is not None:

   # get XML from selected book
   document = implementation.createDocument( None, None, None )
   document.appendChild( session[ "bookToAdd" ].getXML(
      document ) )

   # process XML against XSLT stylesheet
   processor = Processor.Processor()
   style = open( session[ "agent" ] + "/displayBook.xsl" )
   processor.appendStylesheetString( style.read() % \
      ( session[ "ID" ], session[ "ID" ] ) )
   results = processor.runNode( document )
   style.close()

   # display content type and processed XML
   print session[ "content type" ] + results
```

Fig. 23.11 **displayBook.py** converts the XML representation of the selected book to a client-specific format using an XSLT stylesheet (part 2 of 3).

```
55       session.saveSession()    # save Session data
56    else:
57
58       # invalid ISBN has been specified
59       Session.redirect( "error.py?message=Nonexistant+ISBN" )
```

Fig. 23.11 **`displayBook.py`** converts the XML representation of the selected book to a client-specific format using an XSLT stylesheet (part 3 of 3).

If the ISBN has not been specified, the user is forwarded to **`error.py`** (line 17). Otherwise, **`displayBook`** loads the session. If successful, **`displayBook`** obtains the list of **`Book`**s from variable **`session`** (line 27). Line 28 sets the **`session`** dictionary key **`bookToAdd`** to value **`None`**, indicating that the specified ISBN has not yet been found in the list of **`Book`**s stored in variable **`titles`**.

Lines 31–35 iterate over **`titles`**, searching for a **`Book`** with the correct ISBN (specified in the query string). If a book is found that has the specified ISBN, **`session`** attribute **`bookToAdd`** is set to the matching **`Book`** object and the loop terminates.

Line 38 checks whether a matching book has been found. If not, the user is redirected to **`error.py`** (line 59). Otherwise, lines 41–55 execute. Line 41 creates a new XML **`Document`**. Lines 42–43 append the **`product`** element of the matching **`Book`** to the **`Document`**, using the **`appendChild`** method. Lines 46–51 process the XML **`Document`** against a client-specific XSLT stylesheet called **`displayBook.xsl`**. The correct stylesheet resides in the subfolder of the current directory named after the client type. Note that we must format the stylesheet, inserting the session ID, before processing. We then display the results to the client and save the session (lines 54–55).

Figure 23.12 contains the **`displayBook.xsl`** style sheet file used in the XSLT transformation. The values of six elements in the XML document are placed in the resulting XHTML document. The resulting XHTML document is shown in the screen capture at the end of Fig. 23.12.

```
<?xml version = "1.0"?>
<!-- Fig. 23.11: displayBook.xsl                        -->
<!-- XSLT stylesheet that transforms XML generated by   -->
<!-- displayBook.py into XHTML.                         -->
<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <xsl:output method = "xml" omit-xml-declaration = "no"
      indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
      doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>

   <!-- specify the root of the XML document -->
   <!-- that references this stylesheet      -->
   <xsl:template match = "product">
      <html xmlns = "http://www.w3.org/1999/xhtml"
         xml:lang = "en" lang = "en">

      <head>
```

Fig. 23.12 XSLT stylesheet that transforms a book's XML representation into an XHTML document.

```

         <!-- obtain book title from script to place in title -->
         <title><xsl:value-of select = "title"/></title>

         <link rel = "stylesheet" href = "/bookstore/styles.css"
            type = "text/css" />
      </head>

      <body>
         <p class = "bigFont"><xsl:value-of select = "title"/></p>

         <table>
            <tr>
               <!-- create table cell for product image -->
               <td rowspan = "5">  <!-- cell spans 5 rows -->
                  <img src = "/bookstore/images/{ imageFile }"
                     alt = "{ title }" />
               </td>

               <!-- create table cells for price in row 1 -->
               <td class = "bold">Price:</td>

               <td><xsl:value-of select = "price"/></td>
            </tr>

            <tr>

               <!-- create table cells for ISBN in row 2 -->
               <td class = "bold">ISBN #:</td>

               <td><xsl:value-of select = "isbn"/></td>
            </tr>

            <tr>

               <!-- create table cells for edition in row 3 -->
               <td class = "bold">Edition:</td>

               <td><xsl:value-of select = "editionNumber"/></td>
            </tr>

            <tr>

               <!-- create table cells for copyright in row 4 -->
               <td class = "bold">Copyright:</td>

               <td><xsl:value-of select = "copyright"/></td>
            </tr>

            <tr>

               <!-- create Add to Cart button in row 5 -->
               <td>
```

Fig. 23.12 XSLT stylesheet that transforms a book's XML representation into an XHTML document.

```
72                  <form method = "post" action = "addToCart.py?ID=%s">
73                        <p><input type = "submit"
74                              value = "Add to Cart"/></p>
75                     </form>
76                  </td>
77
78                  <!-- create View Cart button in row 5 -->
79                  <td>
80                   <form method = "post" action = "viewCart.py?ID=%s">
81                    <p><input type = "submit" value = "View Cart"/></p>
82                     </form>
83                  </td>
84               </tr>
85            </table>
86
87         </body>
88
89         </html>
90
91      </xsl:template>
92   </xsl:stylesheet>
```

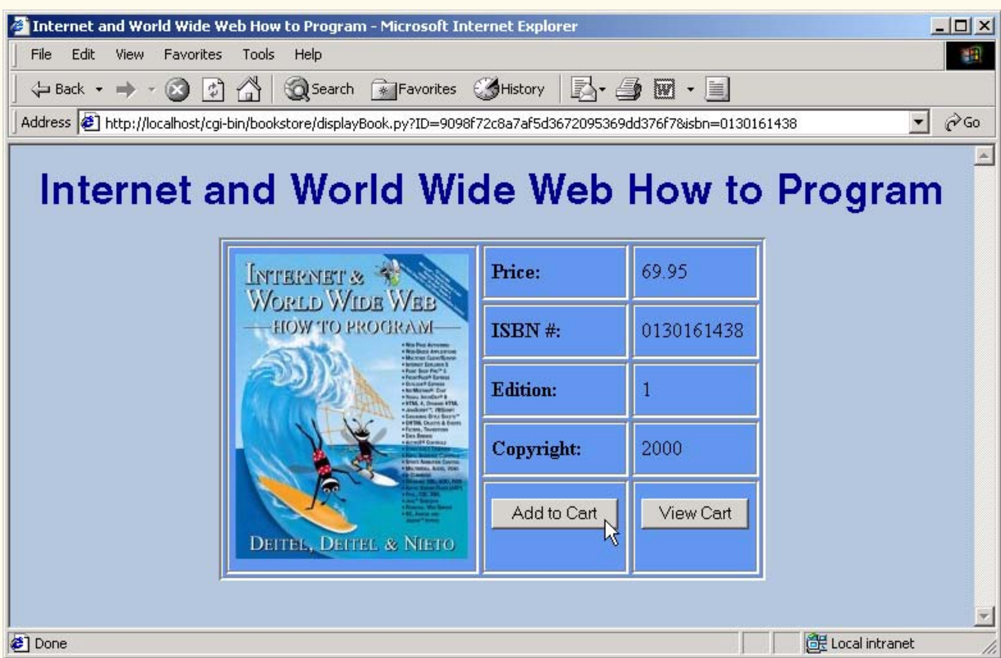


Fig. 23.12 XSLT stylesheet that transforms a book’s XML representation into an XHTML document.

Lines 21 and 28 place the book’s **`title`** in the document’s **`title`** element and in a paragraph at the beginning of the document’s **`body`** element, respectively. Line 34 specifies an **`img`** element that holds the value of the **`imageFile`** element of an XML docu-

ment. This element specifies the name of the file representing the book's cover image. Line 35 specifies the **alt** attribute of the **img** element using the book's **title**. Lines 41, 49, 57 and 65 place the book's **price**, **isbn**, **editionNumber** and **copyright** in table cells, respectively. Lines 72–75 and lines 80-82 create **Add to Cart** (**addToCart.py**) and **View Cart** (**viewCart.py**) buttons, respectively. Both buttons use the **POST** form method to pass the session ID to their target file.

23.9 Adding an Item to the Shopping Cart

When the user presses the **Add to Cart** button in the document produced by the last section, the **addToCart.py** program updates the shopping cart. Items in the shopping cart are represented with **CartItem** objects. An instance of this class maintains an item and the current quantity for that item in the shopping cart. For use with our online bookstore, **CartItem**s maintains a **Book** object and the quantity of that **Book** in the cart. When the user adds an item to the cart, if that **Book** already is represented in the cart with a **CartItem**, the quantity of that item is updated in the class. Otherwise, the script creates a new **CartItem** with a quantity of 1. After updating the cart, the user is forwarded to **viewCart.py** to view the current cart contents. Class **CartItem** and **addToCart.py** are shown in Fig. 23.13 and Fig. 23.14, respectively.

```
# Fig. 23.12: CartItem.py
# Maintains an item and a quantity

class CartItem:
   """Class that maintains an item and its quantity"""

   def __init__( self, itemToAdd, number ):
      """Initialize a CartItem"""

      self.item = itemToAdd
      self.quantity = number

   def getItem( self ):
      """Get the item"""

      return self.item

   def setQuantity( self, number ):
      """Set the quantity"""

      self.quantity = number

   def getQuantity( self ):
      """Get the quantity"""

      return self.quantity
```

Fig. 23.13 **CartItem**s contain an item and the **quantity** of an item in the shopping cart.

```
1  #!c:\Python\python.exe
2  # Fig. 23.13: addToCart.py
3  # Create new/update CartItem for selected Book object
4
5  import sys
6  import Session
7  import CartItem
8
9  # load Session
10 try:
11    session = Session.Session()
12 except Session.SessionError, message:   # invalid/no session ID
13    Session.redirect( "error.py?message=%s" % message )
14    sys.exit()
15
16 book = session[ "bookToAdd" ]
17 bookISBN = book.getISBN()
18 cart = session[ "cart" ]
19 alreadyInCart = 0        # book has not been found in cart
20
21 # determine if book is in cart
22 for isbn in cart.keys():
23
24    if isbn == bookISBN:
25       alreadyInCart = 1
26       cartItem = cart[ isbn ]
27       break
28
29 # if book is already in cart, update quantity
30 if alreadyInCart:
31    cartItem.setQuantity( cartItem.getQuantity() + 1 )
32
33 # otherwise, create and add a new CartItem to cart
34 else:
35    cart[ book.getISBN() ] = CartItem.CartItem( book, 1 )
36
37 # update cart attribute
38 session[ "cart" ] = cart
39
40 # send user to viewCart.py
41 nextPage = "viewCart.py?ID=%s" % session[ "ID" ]
42 session.saveSession()   # save Session data
43 Session.redirect( nextPage )
```

Fig. 23.14 **addToCart.py** places an item in the shopping cart and invokes **viewCart.py** to display the cart contents.

The program first obtains the **Session** object for the current client (lines 10–14). If a session does not exist for this client, a **RequestDispatcher** forwards the request to **error.py** (line 13). Otherwise, line 16 obtains the value of session attribute **bookToAdd**—the **Book** representing the book to add to the shopping cart. Lines 17 obtains this **Book**'s ISBN. Line 18 obtains the value of session attribute **cart**—the dictionary that represents the shopping cart. Lines 22–27 locate the **CartItem** for the book being added

to the cart. If the shopping cart already contains an item for the specified book, line 31 increments the quantity for that **CartItem**. Otherwise, line 35 creates a new **CartItem** with a quantity of 1 and puts the item into the shopping cart, keyed by the book ISBN. Line 38 sets **cart** session attribute to reference the dictionary **cart**. Then, lines 41-43 forward the user to **viewCart.py** to display the cart contents.

23.10 Viewing the Shopping Cart

Program **viewCart.py** (Fig. 23.15) extracts the **CartItem**s from the shopping cart, subtotals each item in the cart, totals all the items in the cart and creates a document that allows the client to view the cart in tabular format.

```
#!c:\Python\python.exe
# Fig. 23.14: viewCart.py
# Generate XML representing cart, convert
# to required format using client-specific XSLT
# stylesheet and display results.

import sys
import Session
from xml.xslt import Processor
from xml.dom.DOMImplementation import implementation

# load Session
try:
   session = Session.Session()
except Session.SessionError, message:   # invalid/no session ID
   Session.redirect( "error.py?message=%s" % message )
   sys.exit()

cart = session[ "cart" ]
total = 0              # total for all ordered items

# generate XML representing cart object
document = implementation.createDocument( None, None, None )
cartNode = document.createElement( "cart" )
document.appendChild( cartNode )

# add XML representation for each cart item
for item in cart.values():

   # get book data, calculate subtotal and total
   book = item.getItem()
   quantity = item.getQuantity()
   price = float( book.getPrice() )
   subtotal = quantity * price
   total += subtotal

   # create an orderProduct element
   orderProduct = document.createElement( "orderProduct" )
```

Fig. 23.15 **viewCart.py** obtains the shopping cart and outputs a document with the cart contents in tabular format.

```
39
40        # create a product element and append to orderProduct
41        productNode = book.getXML( document )
42        orderProduct.appendChild( productNode )
43
44        # create a quantity element and append to orderProduct
45        quantityNode = document.createElement( "quantity" )
46        quantityNode.appendChild( document.createTextNode( "%d" %
47           quantity ) )
48        orderProduct.appendChild( quantityNode )
49
50        # create a subtotal element and append to orderProduct
51        subtotalNode = document.createElement( "subtotal" )
52        subtotalNode.appendChild( document.createTextNode( "%.2f" %
53           subtotal ) )
54        orderProduct.appendChild( subtotalNode )
55
56        # append orderProduct to cartNode
57        cartNode.appendChild( orderProduct )
58
59     # set the total attribute of cart element
60     cartNode.setAttribute( "total", "%.2f" % total )
61
62     # make current total a session attribute
63     session[ "total" ] = total
64
65     # process generated XML against XSLT stylesheet
66     processor = Processor.Processor()
67     style = open( session[ "agent" ] + "/viewCart.xsl" )
68     processor.appendStylesheetString( style.read() % ( session[ "ID" ],
69        session[ "ID" ] ) )
70     results = processor.runNode( document )
71     style.close()
72
73     # display content type and processed XML
74     pageData = session[ "content type" ] + results
75     session.saveSession()   # save Session data
76     print pageData
```

Fig. 23.15 **viewCart.py** obtains the shopping cart and outputs a document with the cart contents in tabular format.

We first load the session (lines 13–17). If an error occurs, the client is redirected to **error.py**. Line 19 obtains the shopping cart attribute of the session. We then create a new XML **Document** and append a **cart** element to **Document** (lines 23–25).

Lines 28–57 compute the total of the items in the cart. Lines 31, 32 and 33 retrieve the **Book** object, the quantity and the price from the **CartItem**, respectively. Line 34 calculates the subtotal for the **CartItem**. Line 35 updates the total cost of all cart items. Line 38 creates an XML **orderProduct** element for each item in the cart.

Each **orderProduct** element contains 3 children elements: **product**, **quantity** and **subtotal**. We first retrieve and append the **product** child of **orderProduct** (lines 41–42). Lines 45–48 then create and append the **quantity** element. Note that the quantity of the current **CartItem** must be formatted to a string before creating the ele-

ment. Lines 51-54 create and append the **subtotal** child of **orderProduct**. The **subtotal** element contains the subtotal of the current **CartItem**, formatted to two decimal places. Line 57 appends the current **orderProduct** to the **cart** element.

When an **orderProduct** element has been created and appended to the **cart** element for each **CartItem**, the **total** attribute of the **cart** element is then set (line 60). Line 63 stores the current sales total in the **total** attribute of the session. Lines 66–71 process the XML **Document** against a client-specific XSLT stylesheet (**viewCart.xsl**). Note that session ID must once again be inserted into the stylesheet before processing. Lines 74–76 save the session and display the translated XML to the client.

Figure 23.16 contains the **viewCart.xsl** style sheet file used in the XSLT transformation for an html client. The resulting XHTML document is shown in the screen capture at the end of Fig. 23.16.

```
<?xml version = "1.0"?>
<!-- Fig. 23.15: viewCart.xsl                              -->
<!-- XSLT stylesheet that transforms XML generated by  -->
<!-- viewCart.py into XHTML.                               -->
<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <xsl:output method = "xml" omit-xml-declaration = "no"
      indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
      doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>

   <xsl:template match = "cart">
      <html xmlns = "http://www.w3.org/1999/xhtml"
            xml:lang = "en" lang = "en">

      <head>
         <title>Your Online Shopping Cart</title>
         <link rel = "stylesheet" href = "/bookstore/styles.css"
            type = "text/css" />
      </head>

      <body>

         <p class = "bigFont">Shopping Cart</p>

         <xsl:choose>
            <xsl:when test = "@total = '0.00'">
               <p class = "bold">
                  Your shopping cart is currently empty.</p>
            </xsl:when>

            <xsl:otherwise>  <!-- total != 0.00 -->
               <table class = "cart">
                  <tr>
                     <th>Product</th>
                     <th>Quantity</th>
                     <th>Price</th>
```

Fig. 23.16 XSLT stylesheet that transforms a cart's XML representation into an XHTML document.

```
                        <th>Total</th>
                     </tr>

                     <xsl:apply-templates select = "orderProduct">

                        <!-- sort orderProducts by product/title -->
                        <xsl:sort select = "product/title"/>

                     </xsl:apply-templates>

                     <tr>
                        <td colspan = "4"
                           class = "bold right">Total: <xsl:value-of
                           select = "@total"/></td>
                     </tr>
                  </table>

               </xsl:otherwise>
            </xsl:choose>

            <p class = "bold green">
               <a href = "allBooks.py?ID=%s">Continue Shopping</a>
            </p>

            <form method = "post" action = "order.py?ID=%s">
               <p><input type = "submit" value = "Check Out" /></p>
            </form>

         </body>
         </html>
      </xsl:template>

      <xsl:template match = "orderProduct">

         <tr>
            <td><xsl:value-of select = "product/title"/>,
               <xsl:value-of select = "product/editionNumber"/>e</td>
            <td><xsl:value-of select = "quantity"/></td>
            <td class = "right"><xsl:value-of select =
               "product/price"/></td>
            <td class = "bold right"><xsl:value-of select =
               "subtotal"/></td>
         </tr>

      </xsl:template>
   </xsl:stylesheet>
```

Fig. 23.16 XSLT stylesheet that transforms a cart's XML representation into an XHTML document.

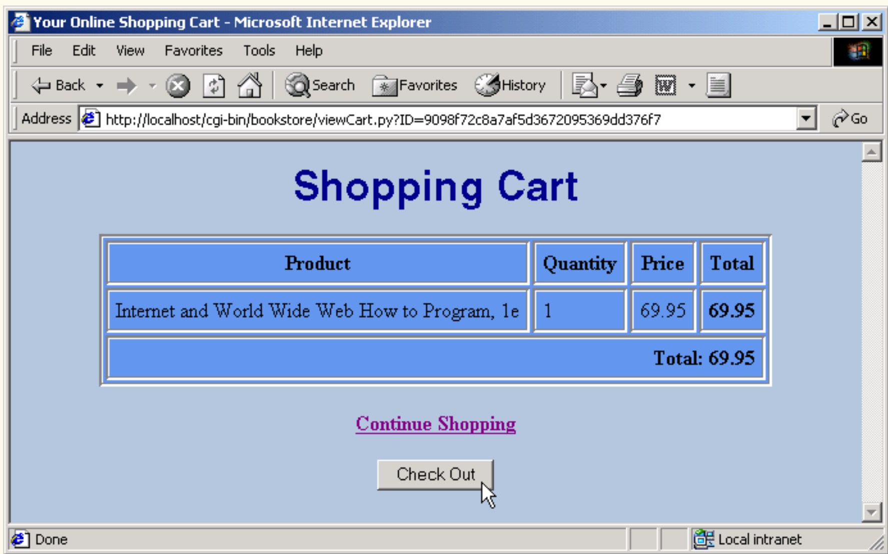


Fig. 23.16 XSLT stylesheet that transforms a cart's XML representation into an XHTML document.

The first **xsl:template** (lines 12–68) matches **cart** elements. Line 26 begins an **xsl:choose** element. If **cart** attribute (denoted by **@**) **total** is equal to **"0.00"**, lines 28–29 execute. Lines 28 and 29 display a message to the client indicating the shopping cart is currently empty. If, however, **total** is not **"0.00"**, lines 33-54 are executed, creating a table for all the items in the cart.

Lines 41–46 insert all matches to the **orderProduct** template, sorted by their **product/title** element. The **orderProduct** template (lines 70–82) matches **orderProduct** elements. Lines 73–79 insert the **orderProduct**'s **product/title**, **product/editionNumber**, **quantity**, **product/price** and **subtotal** in table cells. Lines 49–51 then insert a table row displaying the total for all items. We then create two options for the user. The first is a hyperlink that points to **allBooks.py** (line 59). The second is a **Check Out** button that takes the user to **order.py** (lines 62-64).

23.11 Checking Out

When viewing the cart, the user can click a **Check Out** button to proceed to **order.py** (Fig. 23.17). This script retrieves a static page called **orderForm** which is different for each client type. The correct file is stored in a subdirectory named after the client type (e.g. **orderForm.html** for an HTML client). File **orderForm** is a form in which the user inputs name, address, and credit card information to complete an order. In this example, the form has no functionality. However, it is provided to help complete the application. Normally, there would be some client-side validation of the form elements, some server-side

validation of form elements or a combination of both. When the user presses the button, the browser requests **process.py** to finalize the book order.

```
1  #!c:\Python\python.exe
2  # Fig. 23.16: order.py
3  # Display order form to get information from customer
4
5  import sys
6  import Session
7
8  # load Session
9  try:
10    session = Session.Session()
11 except Session.SessionError, message:   # invalid/no session ID
12    Session.redirect( "error.py?message=%s" % message )
13    sys.exit()
14
15 # display content type and orderForm for specific client-type
16 content = open( "%s/orderForm.%s" % ( session[ "agent" ],
17    session[ "extension" ] ) )
18 pageData = session[ "content type" ] + content.read() % \
19    session[ "ID" ]
20 content.close()
21
22 session.saveSession()   # save Session data
23 print pageData
```

Fig. 23.17 **order.py** retrieves, formats and displays a static order form page to the client.

Lines 9–13 first load the session. If an error occurs, the client is forwarded to **error.py**. Line 16 opens the client-specific order form. Note that for convenience, the directory name is the same as the file extension. Lines 18–19 create the string that contains the client content type and the contents of **orderForm**, formatted with the session ID. The session then saves and the order form displays (lines 22–23).

Figure 23.18 shows **orderForm.html**, the order form displayed by **order.py** to HTML clients. The resulting XHTML document is displayed in the screenshot below.

```
<!-- Fig. 23.17: orderForm.html                   -->
<!-- Static XHTML to be displayed by order.py     -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml"
   xml:lang = "en" lang = "en">

<head>

   <title>Order</title>
```

Fig. 23.18 **orderForm.html** is the order form displayed by **order.py** for html clients.

```

   <link rel = "stylesheet" href = "/bookstore/styles.css"
      type = "text/css" />

</head>

<body>
   <p class = "bigFont">Shopping Cart Check Out</p>

   <!-- Form to input user information and credit card. -->
   <!-- Note: No need to input real data in this example. -->
   <form method = "post" action = "process.py?ID=%s">
      <p style = "font-weight: bold">
         Please input the following information</p>

      <table>
         <tr>
            <td class = "right bold">First name:</td>

            <td>
               <input type = "text" name = "firstname"
                  size = "25" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">Last name:</td>

            <td>
               <input type = "text" name = "lastname"
                  size = "25" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">Street:</td>

            <td>
               <input type = "text" name = "street"
                  size = "25" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">City:</td>

            <td>
               <input type = "text" name = "city"
                  size = "25" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">State:</td>

            <td>
               <input type = "text" name = "state"
```

Fig. 23.18 **orderForm.html** is the order form displayed by **order.py** for html clients.

```
                  size = "2" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">Zip code:</td>

            <td>
               <input type = "text" name = "zipcode"
                  size = "10" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">Phone #:</td>

            <td>
               (
                  <input type = "text" name = "phone" size = "3" />
               )

              <input type = "text" name = "phone2" size = "3" /> -

               <input type = "text" name = "phone3" size = "4" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">Credit Card #:</td>

            <td>
               <input type = "text" name = "creditcard"
                  size = "25" />
            </td>
         </tr>
         <tr>
            <td class = "right bold">Expiration (mm/yy):</td>

            <td>
               <input type = "text" name = "expires"
                  size = "2" />

               <input type = "text" name = "expires2"
                  size = "2" />
            </td>
         </tr>
      </table>

      <p><input type = "submit" value = "Submit" /></p>
   </form>
</body>

</html>
```

Fig. 23.18 **orderForm.html** is the order form displayed by **order.py** for html clients.

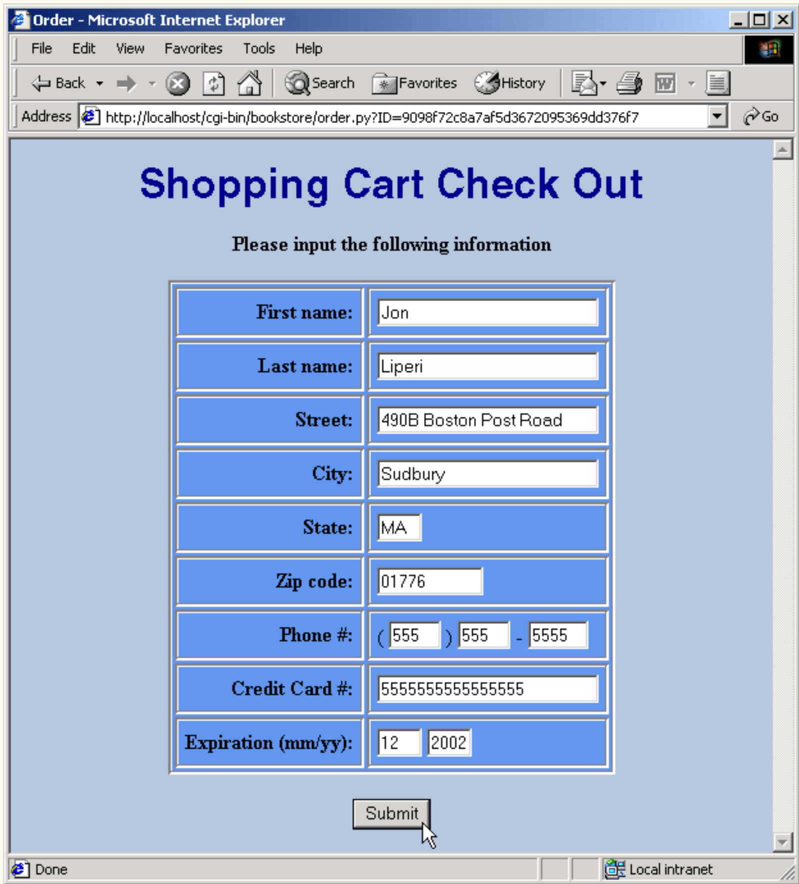


Fig. 23.18 **`orderForm.html`** is the order form displayed by **`order.py`** for html clients.

23.12 Processing the Order

Figure 23.19 (**`process.py`**) pretends to process the user's credit-card information and retrieves a client-specific document called **`thankYou`** (lines 16–17)—a static client-specific page. The correct file is stored in a subdirectory named after the client type (e.g., **`thankYou.html`** for an HTML client). The program then inserts the final dollar total into the contents of **`thankYou`** (lines 18–19), and displays this page for the client. Our simulation of a bookstore does not perform real credit-card processing, so the transaction is now complete. Line 23 invokes **`Session`** method **`delete`** to discard the session object for the current client. In a real store, the session would not be invalidated until the purchase is con-

firmed by the credit-card company. Figure 23.20 shows file **thankYou** for an HTML client. The resulting XHTML document is displayed in the screenshot below.

```python
#!c:\Python\python.exe
# Fig. 23.18: process.py
# Display thank you page to customer and delete session

import sys
import Session

# load session
try:
   session = Session.Session()
except Session.SessionError, message:   # invalid/no session ID
   Session.redirect( "error.py?message=%s" % message )
   sys.exit()

# display content type and thankYou for specific client-type
content = open( "%s/thankYou.%s" % ( session[ "agent" ],
   session[ "extension" ] ) )
pageData = session[ "content type" ] + content.read() % \
   session[ "total" ]
content.close()

# delete session because processing is complete
session.deleteSession()
print pageData
```

Fig. 23.19 **process.py** retrieves, formats and displays a static thank you page to the client.

```html
<!-- thankYou.html                                  -->
<!-- Static XHTML to be displayed by process.py    -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml"
   xml:lang = "en" lang = "en">

<head>

   <title>Thank You!</title>

   <link rel = "stylesheet" href = "/bookstore/styles.css"
      type = "text/css" />

</head>

<body>
   <p class = "bigFont">Thank You</p>
   <p>Your order has been processed.</p>
```

Fig. 23.20 **thankYou.html** is the exit page displayed by **process.py** for HTML clients.

```
    <p>Your credit card has been billed:
       <span class = "bold">$%.2f</span>
    </p>
 </body>

 </html>
```

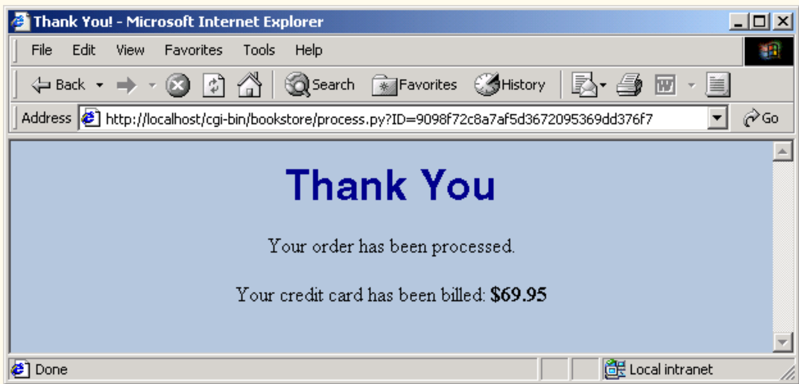


Fig. 23.20 **thankYou.html** is the exit page displayed by **process.py** for HTML clients.

23.13 Error Handling

When an error occurs in our online bookstore, the client is forwarded to **error.py** (Fig. 23.21).

If an error message is specified in the query string, we begin by creating a new XML **Document** (line 16). Lines 17–18 create **error** and **message** elements, respectively. Lines 19–20 append the specified error message to the **message** element. Line 21 appends the **message** element to the **error** element. Line 22 appends the **error** element to the XML **Document**.

```
1  #!c:\Python\python.exe
2  # Fig. 23.20: error.py
3  # Generate XML error message and display to user
4  # using  client-specific XSLT stylesheet.
5
6  import cgi
7  import Session
8  from xml.xslt import Processor
9  from xml.dom.DOMImplementation import implementation
10
11 form = cgi.FieldStorage()
12
13 if form.has_key( "message" ):
14
```

Fig. 23.21 **error.py** displays a dynamically created error page.

```
15    # create DOM for error message
16    document = implementation.createDocument( None, None, None )
17    error = document.createElement( "error" )
18    message = document.createElement( "message" )
19    message.appendChild( document.createTextNode(
20       form[ "message" ].value ) )
21    error.appendChild( message )
22    document.appendChild( error )
23
24    # process against XSLT stylesheet
25    processor = Processor.Processor()
26    style = open( Session.getClientType()[ 0 ] + "/error.xsl" )
27    processor.appendStylesheetStream( style )
28    results = processor.runNode( document )
29    style.close()
30
31    # display content type and processed XML
32    print Session.getContentType() + results
```

Fig. 23.21 **error.py** displays a dynamically created error page.

Lines 25–29 process the **Document** against a client-specific XSLT stylesheet (**error.xsl**). Note that because **error.py** has no session, it must call **Session** functions **getClientType** and **getContentType**, to determine the correct files to use. The results are displayed for the user (line 32).

Figure 23.22 contains the **error.xsl** style sheet file used in the XSLT transformation for an HTML client. Lines 12–31 define an **xsl:template** which matches **error** elements. Line 26 inserts the value of the **message** element into a **paragraph** tag. The resulting XHTML document is shown in the screen capture at the end of Fig. 23.22.

```
1  <?xml version = "1.0"?>
2  <!-- Fig. 23.21: error.xsl                              -->
3  <!-- XSLT stylesheet that transforms XML generated by   -->
4  <!-- error.py into XHTML.                               -->
5  <xsl:stylesheet version = "1.0"
6     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
7
8     <xsl:output method = "xml" omit-xml-declaration = "no"
9        indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
10       doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
11
12    <xsl:template match = "error">
13       <html xmlns = "http://www.w3.org/1999/xhtml"
14          xml:lang = "en" lang = "en">
15          <head>
16             <title>Error</title>
17            <link rel = "stylesheet" href = "/bookstore/styles.css"
18                type = "text/css" />
19          </head>
20
```

Fig. 23.22 XSLT stylesheet that transforms the XML representation of an error into an XHTML document.

```
         <body>

         <p class = "bigFont">Error message:</p>

         <p class = "bold">
            <xsl:value-of select = "message"/>
         </p>

      </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

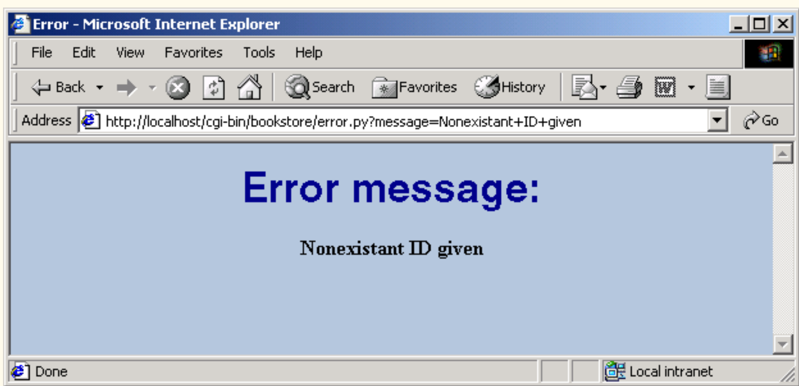


Fig. 23.22 XSLT stylesheet that transforms the XML representation of an error into an XHTML document.

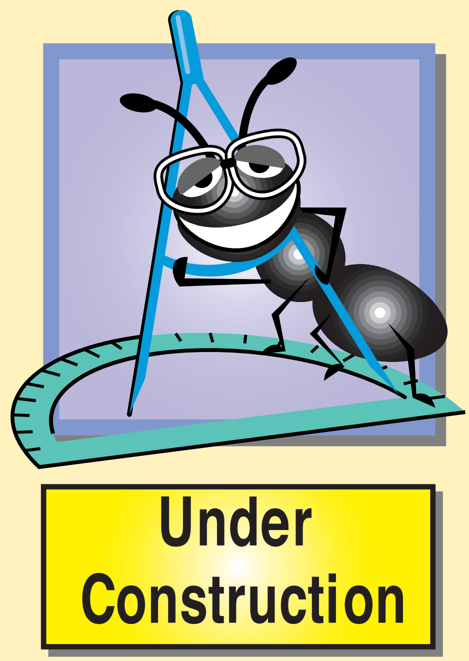
23.14 Handling Wireless Clients (XHTML Basic and WML)

24

Multimedia

Objective

- To introduce multimedia applications in Python.
- To understand how to create 3D objects with module **`PyOpenGL`**.
- To manipulate Alice 3D objects .
- To create a CD player with module **`pygame`**.
- To use module **`pygame`** to create a 2D Space Cruiser game.



Outline

24.1 Introduction

In addition to its many other capabilities, Python allows programmers to create interactive multimedia applications. It is increasingly important for programmers to be able to create multimedia components. We provide examples using **PyOpenGL** and Alice.

24.2 Introduction to PyOpenGL

Module ***PyOpenGL*** is a wrapper for *OpenGL*. OpenGL is a language for rendering 3D graphics. The **PyOpenGL** module allows the programmer to write Python programs that create colorful, interactive 3D graphics.

OpenGL needs a context in which all its rendering can be displayed. GLUT, wxPython and FxPy are possible contexts. The examples in this chapter use **Tkinter** as OpenGL's context.

Module **PyOpenGL** includes the **Tkinter** component **Opengl** which allows openGL to be displayed. There are two other components in which openGL can be displayed—**BaseOpengl**, from which **Opengl** inherits, and **Pogl**.

By default, the **Opengl** component has an event bound to each mouse button. Holding the left mouse button allows the user to move objects in the **Opengl** component. The middle mouse button rotates objects and the right mouse button resizes objects.

24.3 PyOpenGL examples

In this section we present two **PyOpenGL** examples. Figure 24.1 uses **PyOpenGL** coloring and transformations to create a rotating, colored box. Note that because we are using

Tkinter as the OpenGL context, the program structure is similar to programs found in Chapters 10 and 11.

```
#!c:\Python\python.exe
# A colored, rotating box (with open top and bottom)

from Tkinter import *
from OpenGL.GL import *
from OpenGL.Tk import *

class ColorBox( Frame ):
   """A colored, rotating box"""

   def __init__( self ):
      """Initialize GUI and OpenGL"""

      Frame.__init__( self )
      self.master.title( "Color Box" )
      self.master.geometry( "300x300" )
      self.pack( expand = YES, fill = BOTH )

      # create and pack Opengl -- use double buffering
      self.openGL = Opengl( self, double = 1 )
      self.openGL.pack( expand = YES, fill = BOTH )

      self.openGL.redraw = self.redraw     # set redraw function
      self.openGL.set_eyepoint( 20 )       # move away from object

      self.amountRotated = 0   # alternate rotating left/right
      self.increment = 2       # rotate amount
      self.update()            # begin rotation

   def redraw( self, openGL ):
      """Draw box on black background"""

      # clear background and disable lighting
      glClearColor( 0.0, 0.0, 0.0, 0.0 )
      glClear( GL_COLOR_BUFFER_BIT )   # select clear color
      glDisable( GL_LIGHTING )         # paint background

      # constants
      red = ( 1.0, 0.0, 0.0 )
      green = ( 0.0, 1.0, 0.0 )
      blue = ( 0.0, 0.0, 1.0 )
      purple = ( 1.0, 0.0, 1.0 )

      vertices = \
         [ ( ( -3.0, 3.0, -3.0 ), red ),
           ( ( -3.0, -3.0, -3.0 ), green ),
           ( ( 3.0, 3.0, -3.0 ), blue ),
           ( ( 3.0, -3.0, -3.0 ), purple ),
           ( ( 3.0, 3.0, 3.0 ), red ),
           ( ( 3.0, -3.0, 3.0 ), green ),
```

Fig. 24.1 Using **Opengl** with **Tkinter** context.

```
           ( ( -3.0, 3.0, 3.0 ), blue ),
           ( ( -3.0, -3.0, 3.0 ), purple ),
           ( ( -3.0, 3.0, -3.0 ), red ),
           ( ( -3.0, -3.0, -3.0 ), green ) ]

      glBegin( GL_QUAD_STRIP )    # being drawing

      # change color and plot point for each vertex
      for vertex in vertices:
         location, color = vertex
         apply( glColor3f, color )
         apply( glVertex3f, location )

      glEnd()                     # stop drawing
      glEnable( GL_LIGHTING )     # re-enable lighting

   def update( self ):
      """Rotate box"""

      if self.amountRotated >= 500:   # change rotation direction
         self.increment = -2          # rotate left
      elif self.amountRotated <= 0:   # change rotation direction
         self.increment = 2           # rotate right

      # rotate box around ( 1.0, 1.0, 1.0 )
      glRotate( self.increment, 1.0, 1.0, 1.0 )
      self.amountRotated += self.increment

      self.openGL.tkRedraw()                 # redraw geometry
      self.openGL.after( 10, self.update )   # call update in 10ms

def main():
   ColorBox().mainloop()

if __name__ == "__main__":
   main()
```

Fig. 24.1 Using **Opengl** with **Tkinter** context.

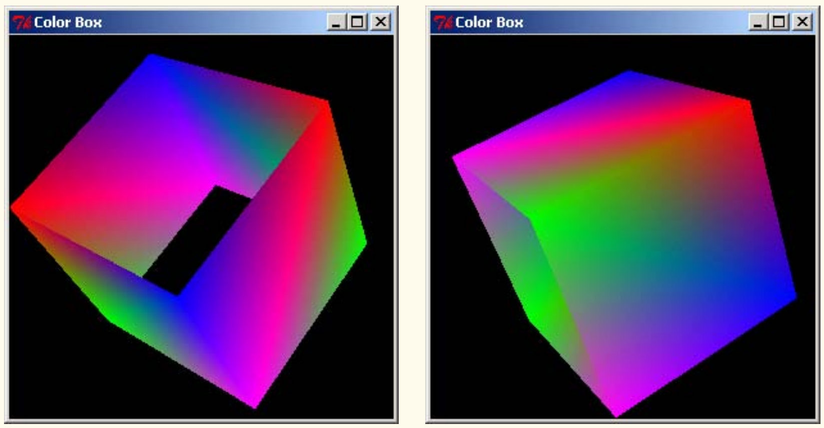


Fig. 24.1 Using **Opengl** with **Tkinter** context.

Line 83 creates an instance of class **ColorBox** (lines 11–80) and enters its **mainloop**. The **ColorBox** constructor (lines 11–28) first initializes the window (lines 14–17). Lines 20–21 create and pack an **Opengl** component—**openGL**—which is used to render the OpenGL objects. **openGL** attribute **double** is set to 1 to ensure that *double buffering* is used. With double buffering, OpenGL maintains two screen buffers—one to display and one to update. When the display is updated, the two buffers are simply switched. This ensures that the user does see the screen being updated (which can cause a choppy display).

Line 23 sets **openGL**'s **redraw** method. This method, **redraw**, will be called when the scene must be redrawn (i.e., something has changed). Method **redraw** (lines 30–65) draws the box on the background. Line 34 calls **PyOpenGL** function **glClearColor** to specify the color which will be used by function **glClear** (line 35). Colors are represented by a three-element tuple or four-element tuple in the form (*R*, *G*, *B*) and (*R*, *G*, *B*, *A*), respectively. *R*, *G*, *B* and *A* stand for red, green, blue and alpha (transparency). Possible values are decimal values between 0.0 (none) and 1.0 (full). By combining different values, different colors are achieved; The representation for black is (0.0, 0.0, 0.0, 0.0). Lines 39–42 define some other colors. Line 35 calls **PyOpenGL** function **glClear** to color the background with the previously selected color—black (line 34). The value passed to **glClear**—**GL_COLOR_BUFFER_BIT**—specifies that the color specified should be used to color the background. Line 36 calls **PyOpenGL** function **glDisable** to disable lighting (**GL_LIGHTING**) for this example.

Lines 44–54 create a list of a vertices which define the box. Each element of the list contains a vertex location and designated color. Lines 56–64 draw the box. Line 56 calls **PyOpenGL** function **glBegin** with argument **GL_QUAD_STRIP**. This ensures that any points defined before a subsequent call to function **glEnd** (line 64) will be connected by a strip of polygons. For other acceptable values, review OpenGL documentation. In **PyOpenGL**, three-dimensional points are defined with function **glVertex3f**. Line 60 obtains the vertex location and color for each vertex. Line 61 uses function **apply** to call **PyOpenGL** function **glColor3f** to change the current drawing color. **glColor3f** takes as arguments three floating-point numbers representing an RGB color. Line 62 then

calls function **glVertex3f** to draw a point in three-dimensional space. The color of the point is the color specified by **glColor3f** (line 61). Because each vertex has a unique color, **PyOpenGL** will interpolate between the colors. Line 64 calls **PyOpenGL** function **glEnd**, ending the **GL_QUAD_STRIP**. Finally, line 65 calls **PyOpenGL** function **Enable** to re-enable lighting.

Line 24 calls **Opengl** method **set_eyepoint**. This method moves the camera away from the scene by a specified amount. Lines 26–27 initialize variables **amountRotated** and **increment**. These values will be used to control the rotation of the box. Finally, line 28 invokes method **update**.

Method **update** (lines 67–80) rotates the box. Lines 70–73 alter the rotational direction, represented by variable **increment**. Method **glRotate** (line 76) accepts four parameters. The first parameter, in this case variable **increment**, sets the angle of rotation. The last three floating-point numbers are the coordinates around which the shape rotates. Line 77 increments variable **amountRotated**, which keeps track of how much the box has been rotated. The call to method **tkRedraw** (line 79) causes the **Opengl** component to be redrawn with the rotated shape. Method **after** (line 80) takes 10 and method update as parameters. As a result, **mainloop** schedules **update** to be called every 10ms.

Figure 24.2 demonstrates several methods of the ***OpenGL.GLUT*** module that create three-dimensional shapes. Module **GLUT** is the GL Utilities toolkit. The example creates a GUI that allows the user to preview colors and shapes.

```
#!c:\Python\python.exe
# Demonstrating various GLUT shapes

from Tkinter import *
import Pmw
from OpenGL.GL import *
from OpenGL.Tk import *
from OpenGL.GLUT import *

class ChooseShape( Frame ):
   """Allow user to preview different shapes and colors"""

   def __init__( self ):
      """Create GUI with MenuBar"""

      Frame.__init__( self )
      Pmw.initialise()
      self.master.title( "Choose a shape and color" )
      self.master.geometry( "300x300" )

      # initialize openGL
      self.openGL = Opengl( double = 1 )    # use double-buffering
      self.openGL.redraw = self.redraw      # set redraw function
      self.openGL.pack( expand = YES, fill = BOTH )
      self.openGL.set_eyepoint( 20 )        # move away from object
      self.openGL.autospin_allowed = 1      # allow auto-spin

      # create and pack MenuBar
```

Fig. 24.2 Creating various shapes with **GLUT**.

```
        self.choices = Pmw.MenuBar( self.openGL )
        self.choices.pack( fill = X )

        self.choices.addmenu( "Shape", None )   # Shape submenu

        # possible shapes and arguments
        self.shapes = { "glutWireCube" : ( 3, ),
                        "glutSolidCube": ( 3, ),
                        "glutWireIcosahedron" : (),
                        "glutSolidIcosahedron" : (),
                        "glutWireCone" : ( 3, 3, 50, 50 ),
                        "glutSolidCone" : ( 3, 3, 50, 50 ),
                        "glutWireTorus" : ( 1, 3, 50, 50 ),
                        "glutSolidTorus" : ( 1, 3, 50, 50 ),
                        "glutWireTeapot" : ( 3, ),
                        "glutSolidTeapot" : ( 3, ) }

        self.selectedShape = StringVar()
        self.selectedShape.set( "glutWireCube" )

        # add radiobutton menu item for each shape
        sortedShapes = self.shapes.keys()
        sortedShapes.sort()   # sort names before adding to menu

        for shape in sortedShapes:
           self.choices.addmenuitem( "Shape", "radiobutton",
              label = shape, variable = self.selectedShape )

        self.choices.addmenu( "Color", None )   # Color submenu

        # possible colors and their values
        self.colors = { "White" : ( 1.0, 1.0, 1.0 ),
                        "Blue" : ( 0.0, 0.0, 1.0 ),
                        "Red" : ( 1.0, 0.0, 0.0 ),
                        "Green" : ( 0.0, 1.0, 0.0 ),
                        "Magenta" : ( 1.0, 0.0, 1.0 ) }

        self.selectedColor = StringVar()
        self.selectedColor.set( "White" )

        # add radiobutton menu item for each color
        for color in self.colors.keys():
           self.choices.addmenuitem( "Color", "radiobutton",
              label = color, variable = self.selectedColor )

     def redraw( self, openGL ):
        """Draw selected shape on black background"""

        # clear background and disable lighting
        glClearColor( 0.0, 0.0, 0.0, 0.0 )
        glClear( GL_COLOR_BUFFER_BIT )
        glDisable( GL_LIGHTING )

        # obtain and set selected color
```

Fig. 24.2 Creating various shapes with **GLUT**.

```
83         color = self.selectedColor.get()
84         apply( glColor3f, self.colors[ color ] )
85
86         # obtain and draw selected shape
87         shape = self.selectedShape.get()
88         apply( eval( shape ), self.shapes[ shape ] )
89
90         glEnable( GL_LIGHTING )   # re-enable lighting
91
92   def main():
93      ChooseShape().mainloop()
94
95   if __name__ == "__main__":
96      main()
```

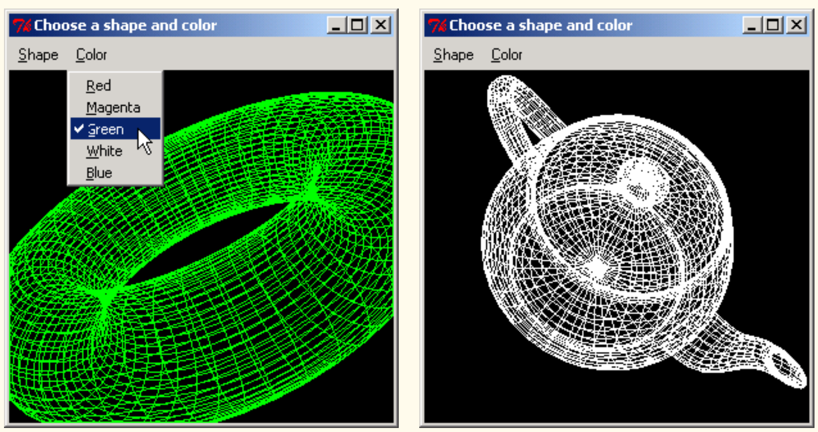


Fig. 24.2 Creating various shapes with **GLUT**.

Line 93 creates an instance of class **ChooseShape** (lines 10–90) and enters its **mainloop**. Lines 22–25 of the constructor create and pack an **Opengl** component in the same way as Fig. 24.1. Line 26 sets **allow_autospin** to 1. As a result, the user can cause a shape to rotate continuously by holding down the middle mouse button, dragging it in the direction of the rotation and releasing it.

Dictionary **shapes** (lines 35–44) contains **GLUT** shapes as its keys. The values are possible arguments to be passed to the methods which are named after the shapes. Methods **glutWireCube** and **glutSolidCube** (lines 35–36) accept the length of the cube's side as a parameter—3 in this case. Methods **glutWireIcosahedron** and **glutSolidIcosahedron** (lines 37–38) accept no parameters and create a 20-sided shape with a radius of 1.0. Methods **glutWireCone** and **glutSolidCone** (lines 39–40) accept four parameters—the base, the height, the number of slices and the number of stacks, i.e., the number of subdivisions of the cone, along the third axis. Methods **glutWireTorus** and **glutSolidTorus** (lines 41–42) accept four parameters as well. The first two specify the inner and outer radii of the doughnut shape. The last two arguments specify the number of sides in each section and the number of divisions in each section. Methods

glutWireTeapot and **glutSolidTeapot** (lines 43–44) accepts the relative size of the teapot shape as a parameter.

Dictionary **colors** (lines 60–64) contains a list of color names as keys. Each color has its RGB tuple as its value. The GUI has a **Pmw MenuBar** with radiobutton menu items for each shape and color. The default selection is a white wire cube.

Method **redraw** (lines 74–90) creates a black background (lines 78–79). Method **get** obtains the selected color and shape. Method **apply** (line 84) applies method **glColor3f** to the RGB value associated with the color key. Method **eval** evaluates the shape function and **apply** applies the method to any arguments associated with the shape's dictionary key.

24.4 Introduction to Alice

Alice (**www.alice.org**) is a 3D Interactive Graphics Programming Environment created by Stage 3 Research Group (**www.alice.org/stage3**). It is designed for use with Microsoft Windows 95/98/NT. Alice makes simple 3D modeling accessible to novice users. The simple scripting language can manipulate 3D objects designed with Teddy2 (**www.mtl.t.u-tokyo.ac.jp/~takeo/teddy/teddy.htm**) modeling software. In addition to the many objects included with Alice, the user can import many common 3D modeling formats (such as .DXF and .OBJ).

Python can control the Alice environment. A simple and intuitive interface allow the user to place objects in the Alice world and to adjust their initial position. After the programmer designs the starting scene, a Python script creates interactive animation in this world. The animations can also be created without any knowledge of Python. The user can access the list of actions for each object with the mouse and can design a sequence of actions. Alice translates the sequence of actions into Python code by Alice. A completed world can be viewed in a browser using a browser plug-in (**www.alice.org/downloads/plugin/**).

Python used with Alice has several significant modifications. In Alice, Python is not case sensitive. As a result, two variables or methods with the same name may not exist in the same namespace. In addition, integer division results in a floating number value (if non-integer). So 1/2 would be equal to 0.5 rather than 0. Usually in Python, division of 1/2 results in 0 because the answer is truncated after the decimal point.

24.5 Fox, Chicken and Seed Problem

Figure 24.3 implements the classical Fox, Chicken and Seed problem as a game. The rules are simple: Alice Liddell needs to transport a fox, a chicken and a seed (a flower pot in this example) across a river with a boat. She has to operate the small boat, which can only accommodate one additional passenger. The problem is that fox will eat the chicken if they

are left on the shore alone. For the same reason, the chicken can not be left alone with the flower.

```
### Everything below this line is hand-edited Python Code ###
# Fig. 24.01: ChickenFoxSeed.py
# Chicken Fox and Seed problem

FollowTheBoat = Loop( camera.PointAt
   ( AliceLiddell.dress.rthigh ) )

# run the two animation together with a given pause time
def AnimateWithPause( Animation1, Animation2, Object, time ):

   return Loop( DoInOrder( DoTogether( Animation1, Animation2 ),
      Wait( time ) ) )

LoopingFish = Loop( AnimateWithPause
   ( Fish.Move( Forward, 50, Duration = 5 ),
     Fish.Turn( Down, 1, Duration = 5 ),
     Fish, 15 ) )
LoopingFish2 = Loop( AnimateWithPause
   ( Fish2.Move( Forward, 70, Duration = 8 ),
     Fish2.Turn( Down, 1, Duration = 8 ),
     Fish2, 25 ) )

# lists that keep track of object position
thisBank = [ "Fox", "Chicken", "Flower" ]
theBoat = []
otherBank = []

currentBank = thisBank
targetBank = otherBank
selected = None

# animal select callback
def animalSelect( value ):

   global selected
   selected = value

# get object into the boat
def ObjectInBoat( Object ):

   Object.RespondToCollisionWith( FishBoat.deck, Object.Stop )
   Object.MoveTo( FishBoat.period )
   Object.Move( Down, 2, Duration = 3 )

# get object out of the boat
def ObjectOutOfBoat( Object ):

   Object.RespondToCollisionWith( Ground, Object.Stop )
   Object.Move( Left, 1 - int( ( len( Object._name ) - 1 ) / 3 ) )
   Object.Move( Back, 7 )
```

Fig. 24.3 Chicken, Fox and Seed (part 1 of 3).

```
   Object.Move( Down, 3, Duration = 3 )


# put the currently selected object into the boat
def getIntoBoat():

   if ( selected in currentBank and
        ( len( theBoat ) == 0 ) and boatArrived() ):
      currentBank.remove( selected )
      theBoat.append( selected )
      ObjectInBoat( eval( selected ) )

# remove currently selected object from the boat.
def getOutOfBoat():

   if ( selected in theBoat and boatArrived() ):
      theBoat.remove( selected )
      currentBank.append( selected )
      ObjectOutOfBoat( eval( selected ) )

# game over, AnimationX defaults to an empty sequence
def finishGame( Animation1, Animation2, final ):

   controlPanel.Destroy()
   FollowTheBoat.stop()
   final.Show()
   DoInOrder( Animation1, Animation2,
   DoInOrder( camera.Place( len( final._name ) + 2,
      InFrontOf, final ), camera.PointAt( final ) ) )

# check if the rules have been violated and the player lost
def checkRules( currentBank ):

   Animation1 = DoInOrder()
   Animation2 = DoInOrder()

   if "Chicken" in currentBank:

      if "Flower" in currentBank:
         Animation1 = DoInOrder( camera.PointAt( Flower ),
            Flower.destroy() )

      if "Fox" in currentBank:
         Animation2 = DoInOrder( camera.PointAt( Chicken ),
            Chicken.destroy() )

      if ( "Flower" in currentBank ) or
           ( "Fox" in currentBank ):
         finishGame( Animation1, Animation2, GAMEOVER )

   if len( currentBank ) == 0 and
        not ( currentBank == targetBank ):
      finishGame( Animation1, Animation2, CONGRATULATIONS )

```

Fig. 24.3 Chicken, Fox and Seed (part 2 of 3).

```
# send the boat to the other shore
def toOtherShore():

   if not boatArrived():   # boat is still in transit
      return

   global currentBank, thisBank, otherBank

   if len( theBoat ) == 1:  # someone is on the boat
      DoInOrder( eval( theBoat[ 0 ] ).Move( Forward, 16,
         Duration = 3 ), eval( theBoat[ 0 ] ).Turn( Left, 1/2, 1,
         AsSeenby = FishBoat ) )

   # move the boat and then set alarm to check rules
   DoInOrder( FishBoat.move( Forward, 16, Duration = 3 ),
      FishBoat.turn( Left, 1/2 ) )
   Alice.SetAlarm( 1, checkRules, ( currentBank ) )

   if currentBank == thisBank:  # switch the currentBank pointer
      currentBank = otherBank
   else:
      currentBank = thisBank

# the boat has arrived
def boatArrived():

   # check to see if the boat is at the shore
   if ( AliceLiddell.DistanceTo( period ) < .01 or
      AliceLiddell.DistanceTo( period2 ) < .01 ):
      return 1
   else:
      return 0

# create the control panel and buttons
controlPanel = AControlPanel ( Caption = "Game Control Panel" )
animalListBox = \
   controlPanel.MakeOptionButtonSet( List = thisBank[ : ],
   Command = animalSelect )
buttonToBoat = \
   controlPanel.MakeButton( Caption = "Get into the boat" )
buttonFromBoat = \
   controlPanel.MakeButton( Caption = "Get out of the boat" )
buttonMoveBoat = \
   controlPanel.MakeButton( Caption = "Go to the other shore" )

buttonToBoat.SetCommand( getIntoBoat )
buttonFromBoat.SetCommand( getOutOfBoat )
buttonMoveBoat.SetCommand( toOtherShore )

# initial selection defaults to the first element (the fox)
animalListbox._children[ 0 ].SetValue( 1 )
selected = "Fox"
```

Fig. 24.3 Chicken, Fox and Seed (part 3 of 3).

The initial scene was created in the Alice world, using predefined objects. Alice Liddell is attached to the boat. The chicken, fox and flower are initially placed next to the boat on the same shore. All other items are inserted merely for decoration. The movements can be controlled using the buttons on **controlPanel** (line 139). The menu allows the user to move the objects in and out of boat and to send Alice across the river. Alice generates the comment in line 1. Code automatically generated by Alice is placed above this comment.

Lines 5–6 continuously point the camera at Alice Liddell. This loop ensures that the **camera** follows Alice Liddell as she moves on the boat. Alice adds a loop to the list of currently running animations and the loop runs until explicitly stopped.

Method **AnimateWithPause** (lines 9–12) combines two animations into one loop. The animations run concurrently and then pause for a given time. This method animates fish movement. Lines 14–21 create the animations for two jumping fish.

Lines 24–29 create lists and initialize them to the starting values. These lists keep track of the objects on the shores and on the boat. Variable **selected** (line 30) holds the currently selected object. Method **animalSelect** (lines 33–36) is a callback for the radio buttons allowing the user to select an object. Method **ObjectInBoat** (lines 39–43) moves a given object into the boat. Line 41 sets **Object.Stop** as the response to collision with the deck of the **FishBoat**. Lines 42–43 move the object above the deck and move it down toward a collision with the deck.

Method **ObjectOutOfBoat** (lines 46–51) moves a given object out of the boat to the shore. The boat movement is symmetric so there is no need to distinguish between the shores when moving the object. Line 48 sets the response to collision with the ground to **Object.Stop**. Line 49 displaces the object based on the name length so that the objects land in different positions on the shore. Line 50–51 move the object back and down accordingly.

Method **getIntoBoat** (lines 55–61) checks whether a currently selected object can be moved into the boat. If the object can be moved into the boat, the method performs the necessary adjustment to the lists. The call **ObjectInBoat.eval(selected)** (line 61) returns the object associated in Python environment with **selected**. Line 57 checks whether **selected** is on the current bank of the river. Line 58 checks whether the boat is empty and whether the animation has finished moving the boat across the river (method **boatArrived**). Lines 59–60 move the selected object from the **currentBank** list to **theBoat** list.

Method **getOutOfBoat** (lines 64–69) checks if a currently selected object can be moved to the shore from the boat. If the object can be moved to the shore, the method performs the necessary adjustment to the lists and calls **ObjectOutOfBoat**. Line 66 checks whether **selected** is in the boat and whether the boat has arrived at the shore. Lines 67–68 move the selected object from list **theBoat** to list **currentBank**.

Method **finishGame** (lines 72–79) cleans up once the player loses or wins the game. Line 74 destroys the **controlPanel**, and line 75 stops the camera animation that follows the boat. Line 76 displays the variable **final**, the result of the game. Lines 77–79 display the two animation parameters and then points the camera at **final**.

Lines 82–103 define method **checkRules**. Lines 84–85 declare empty animations. Lines 87–95 check if one of the rules has been violated and change **Animation1** and **Animation2** accordingly. If one of the conditions has been violated, lines 97–99 call

finishGame with **GAMEOVER** as the result parameter. Lines 101–103 call **finishGame** with **CONGRATULATIONS** as a parameter if all objects were successfully transported to the other shore.

Method **toOtherShore** moves the boat between the river shores. Lines 108–109 returns without changing anything if the boat is in transit. Lines 111 makes global variables accessible within the method. Lines 113–116 checks if there is an object on the boat and if there is, animate that object with the boat. An object on the boat is not a part of the boat, so a separate animation is created to synchronize the object with the boat.

Lines 119–121 create the animation that moves the boat to the other shore. Line 121 sets an alarm so that, a second after the boat leaves the shore, the program checks whether the rules have been violated. Alarms are timed events in Alice. **Alice.SetAlarm** takes the time to wait until setting off an alarm and a function to call at that time. Optionally, parameters for that function can be provided. Lines 123–126 switch the current bank pointer to the other shore.

Method **boatArrived** checks the status of the boat. If the boat is moving across the river, this method returns 0, otherwise it returns 1. This is done using two period objects at two sides of the river. These objects are placed where Alice is located when at the shore and by checking the distance between them we are able to determine if she arrived.

Line 134 creates the **controlPanel** for user input. Lines 141–142 create the set of radio buttons using the list of the objects at the bank and the callback **animalSelect**. Lines 143–146 create the buttons for getting the selected object in and out of the boat. Lines 147–148 create a button that sends the boat across the river. Lines 150–152 set the callbacks

for these buttons. Finally, lines 155–156 set the initial selection to **"Fox"**. Figure 24.4 demonstrates what the example world looks like.



Fig. 24.4 Screenshot of Alice world.

24.6 Introduction to pygame

pygame is a set of Python modules designed for writing games. The pygame modules, written by Pete Shinners, use the *Simple DirectMedia Layer* (*SDL*). SDL is a cross-platform library that provide access to multimedia hardware. **pygame** allows users to access this library through Python. Although various other types of programs have been developed with **pygame**, the most common application is a two-dimensional game. For more information about **pygame**, including extensive documentation, visit **www.pygame.org**.

24.7 Python CD Player

This section demonstrates pygame's **cdrom** module. The **cdrom** module contains class **CD** and functions to initialize the CD-ROM subsystem. Class **CD** represents the user's CD-ROM drive. Methods of this class allow the user to access the CD in the drive. Figure 24.5 creates a simple CD player using **pygame** module **cdrom**. We use **Tkinter** and **Pmw** to create the CD player interface. For more information on **Tkinter** and **Pmw**, review Chapters 10 and 11.

```
#!c:\Python\python.exe
# CDPlayer.py: A simple CD player using Tkinter and pygame

import sys
import string
import pygame, pygame.cdrom
from Tkinter import *
from tkMessageBox import *
import Pmw

class CDPlayer( Frame ):
   """A GUI CDPlayer class using Tkinter and pygame"""

   def __init__( self ):
      """Initialize pygame.cdrom and get CDROM if one exists"""

      pygame.cdrom.init()

      if pygame.cdrom.get_count() > 0:
         self.CD = pygame.cdrom.CD( 0 )
      else:
         sys.exit( "There are no available CDROM drives." )

      self.createGUI()
      self.updateTime()

   def destroy( self ):
      """Stop CD, uninitialize pygame.cdrom and destroy GUI"""

      if self.CD.get_init():
         self.CD.stop()

      pygame.cdrom.quit()
      Frame.destroy( self )

   def createGUI( self ):
      """Create CDPlayer widgets"""

      Frame.__init__( self )
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "CD Player" )

      # display current track playing
```

Fig. 24.5 Python CD player (part 1 of 5).

```
        self.trackLabel = IntVar()
        self.trackLabel.set( 1 )
        self.trackDisplay = Label( self, font = "Courier 14",
           textvariable = self.trackLabel, bg = "black",
           fg = "green" )
        self.trackDisplay.grid( sticky = W+E+N+S )

        # display current time of track playing
        self.timeLabel = StringVar()
        self.timeLabel.set( "00:00/00:00" )
        self.timeDisplay = Label( self, font = "Courier 14",
           textvariable = self.timeLabel, bg = "black",
           fg = "green" )
        self.timeDisplay.grid( row = 0, column = 1, columnspan = 3,
           sticky = W+E+N+S )

        # play/pause CD
        self.playLabel = StringVar()
        self.playLabel.set( "Play" )
        self.play = Button( self, textvariable = self.playLabel,
           command = self.playCD, width = 10 )
        self.play.grid( row = 1, column = 0, columnspan = 2,
           sticky = W+E+N+S )

        # stop CD
        self.stop = Button( self, text = "Stop", width = 10,
           command = self.stopCD )
        self.stop.grid( row = 1, column = 2, columnspan = 2,
           sticky = W+E+N+S )

        # skip to previous track
        self.previous = Button( self, text = "<<<", width = 5,
           command = self.previousTrack )
        self.previous.grid( row = 2, column = 0, sticky = W+E+N+S )

        # skip to next track
        self.next = Button( self, text = ">>>", width = 5,
           command = self.nextTrack )
        self.next.grid( row = 2, column = 1, sticky = W+E+N+S )

        # eject CD
        self.eject = Button( self, text = "Eject", width = 10,
           command = self.ejectCD )
        self.eject.grid( row = 2, column = 2, columnspan = 2,
           sticky = W+E+N+S )

        # pulldown menu of all tracks on CD
        self.trackChoices = Pmw.ComboBox( self, label_text = "Track",
           labelpos = "w", selectioncommand = self.changeTrack,
           fliparrow = 1, listheight = 100 )
        self.trackChoices.grid( row = 3, columnspan = 4,
           sticky = W+E+N+S )

        self.trackChoices.component( "entry" ).config( bg = "grey",
```

Fig. 24.5 Python CD player (part 2 of 5).

```
         fg = "red", state = DISABLED )
      self.trackChoices.component( "listbox" ).config( bg = "grey",
         fg = "red" )

   def playCD( self ):
      """Play/Pause CD if disc is loaded"""

      # if disc has been ejected, reinitialize drive
      if not self.CD.get_init():
         self.CD.init()
         self.currentTrack = 1

         # if no disc in drive, uninitialize and return
         if self.CD.get_empty():
            self.CD.quit()
            return

         # if a disc is loaded, obtain disc information
         else:
            self.totalTracks = self.CD.get_numtracks()
            self.trackChoices.component( "scrolledlist" ).setlist(
               range( 1, self.totalTracks + 1 ) )
            self.trackChoices.selectitem( 0 )

      # if CD is not playing, being play
      if not self.CD.get_busy() and not self.CD.get_paused():
         self.CD.play( self.currentTrack - 1 )
         self.playLabel.set( "| |" )

      # if CD is playing, pause disc
      elif not self.CD.get_paused():
         self.CD.pause()
         self.playLabel.set( "Play" )

      # if CD is paused, resume play
      else:
         self.CD.resume()
         self.playLabel.set( "| |" )

   def stopCD( self ):
      """Stop CD if disc is loaded"""

      if self.CD.get_init():
         self.CD.stop()
         self.playLabel.set( "Play" )

   def playTrack( self, track ):
      """Play track if disc is loaded"""

      if self.CD.get_init():
         self.currentTrack = track
         self.trackLabel.set( self.currentTrack )
         self.trackChoices.selectitem( self.currentTrack - 1 )

```

Fig. 24.5 Python CD player (part 3 of 5).

```
         # start beginning of track
         if self.CD.get_busy():
            self.CD.play( self.currentTrack - 1 )
         elif self.CD.get_paused():
            self.CD.play( self.currentTrack - 1 )
            self.playCD()   # re-pause CD

   def nextTrack( self ):
      """Play next track on CD if disc is loaded"""

      if self.CD.get_init() and \
         self.currentTrack < self.totalTracks:
         self.playTrack( self.currentTrack + 1 )

   def previousTrack( self ):
      """Play previous track on CD if disc is loaded"""

      if self.CD.get_init() and self.currentTrack > 1:
         self.playTrack( self.currentTrack - 1 )

   def changeTrack( self, event ):
      """Play track selected from pulldown menu if disc is loaded"""

      if self.CD.get_init():
         index = int( self.trackChoices.component(
            "scrolledlist" ).curselection()[ 0 ] )
         self.playTrack( index + 1 )

   def ejectCD( self ):
      """Eject CD from drive"""

      response = askyesno( "Eject pushed", "Eject CD?" )

      if response:
         self.CD.init()   # CD must be initialized to eject
         self.CD.eject()
         self.CD.quit()
         self.trackLabel.set( 1 )
         self.timeLabel.set( "00:00/00:00" )
         self.playLabel.set( "Play" )
         self.trackChoices.component( "scrolledlist" ).clear()
         self.trackChoices.component( "entryfield" ).clear()

   def updateTime( self ):
      """Update time display if disc is loaded"""

      if self.CD.get_init():
         seconds = int( self.CD.get_current()[ 1 ] )
         endSeconds = int( self.CD.get_track_length(
            self.currentTrack - 1 ) )

         # if reached end of current track, play next track
         if seconds >= ( endSeconds - 1 ):
            self.nextTrack()
```

Fig. 24.5 Python CD player (part 4 of 5).

```
206             else:
207                minutes = seconds / 60
208                endMinutes = endSeconds / 60
209                seconds = seconds - ( minutes * 60 )
210                endSeconds = endSeconds - ( endMinutes * 60 )
211
212                # display time in format mm:ss/mm:ss
213                trackTime = string.zfill( str( minutes ), 2 ) + \
214                   ":" + string.zfill( str( seconds ), 2 )
215                endTime = string.zfill( str( endMinutes ), 2 ) +  \
216                   ":" + string.zfill( str( endSeconds ), 2 )
217
218                if self.CD.get_paused():
219
220                   # alternate pause symbol and time in display
221                   if not self.timeLabel.get() == "      ||     ":
222                      self.timeLabel.set( "      ||     " )
223                   else:
224                      self.timeLabel.set( trackTime + "/" + endTime )
225
226                else:
227                   self.timeLabel.set( trackTime + "/" + endTime )
228
229          # call updateTime method again after 1000ms ( 1 second )
230          self.after( 1000, self.updateTime )
231
232 def main():
233    CDPlayer().mainloop()
234
235 if __name__ == "__main__":
236     main()
```

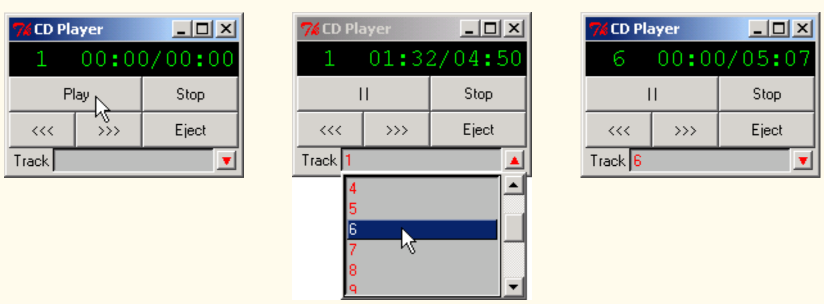


Fig. 24.5 Python CD player (part 5 of 5).

Line 233 creates a **`CDPlayer`** object and enters its **`mainloop`**. The **`CDPlayer`** constructor (lines 14–25) initializes the **`cdrom`** module (line 17). The **`if/else`** statement in lines 19–22 checks to see if there are any available CD-ROM drives by invoking **`cdrom`**'s **`get_count`** function. Function **`get_count`** returns the number of CD-ROMs on the system. If there is at least one CD-ROM, line 20 instantiates a **`CD`** object called **`CD`**. The value passed to the **`CD`** constructor is the ID of the CD-ROM. The program uses the first CD-ROM installed on the system if there is more than one. The constructor receives 0 as

an argument because the first ID is always 0. The program exits (line 22) if no CD-ROM exists.

Line 24 invokes method **createGUI** to create the CD player interface. **createGUI** (lines 36–100) creates various GUI components for the CD player and adds them to the display. Each component's action will be discussed later. Note that the **Label** created to display the track number (**trackDisplay**) and the **Label** created to display the current track time (**timeDisplay**) both have **textvariable**s—**trackLabel** and **timeLabel**—which will be used to update the CD player display. Notice also that **Button play** has a **textvariable**—**playLabel**—which will be used to change its display when the CD player is paused or playing. Lines 91–93 create **trackChoices**, a **Pmw ComboBox** which will be used as a "drop-down" box of track choices. Lines 97–100 use common "mega-widget" method **component** to customize the colors of the drop-down box.

Once the GUI has been created, the constructor calls method **updateTime** (discussed later) and returns, entering the **mainloop**. Once here, the GUI components created can be used.

The **Play** button has callback method **playCD**. **playCD** (lines 102–135) plays or pauses the CD. Line 106 check if the CD-ROM is initialized by invoking **CD** method **get_init**. If the CD-ROM is not initialized, **playCD** initializes it and sets **currentTrack** to 1. **currentTrack** stores the number of the currently playing track. Line 111 checks if the CD-ROM is empty by invoking **CD** method **get_empty**. If the CD-ROM is empty, line 112 uninitializes the CD-ROM with **CD** method **quit** and returns. Otherwise, line 117 obtains the total number of tracks on the disc from **CD** method **get_numtracks** and stores that value in variable **totalTracks**. Lines 118–120 then add them to the drop-down box of track choices (**trackChoices**) and select the first one (track 1).

Line 123 checks if **CD** is not playing and not paused with methods **get_busy** and **get_paused**, respectively. If this is the case, **playCD** invokes **CD** method **play**, specifying what track to play. Note that because tracks numbers for a **CD** object begin with 0 and people generally believe track numbers begin with 1, the value passed to **play** is 1 less than **currentTrack**. Line 125 sets the **Play** button to read **"| |"**, a symbol for **Paused**.

If the CD is playing and not paused, however, lines 129–130 pause the CD (with **CD** method **pause**) and set the **Play** button to read **"Play"** again.

If neither condition is met, however, the CD is paused. If this is the case, lines 134 and 135 resume play with method **resume** and set the **Play** button to read **"| |"** once more. Note that if the CD is currently playing, the **Play** button reads **"| |"**, and if the CD is currently paused, the **Play** button reads **"Play"**.

The **Stop** button has callback **stopCD** (lines 137–142). Line 140 checks if **CD** is initialized. If so, **CD** method **stop** is invoked to stop the CD and the **Play** button is set to read **"Play"** once more. Note that calling **stop** on a **CD** which is not playing does nothing. However, line 140 checks if the CD-ROM is initialized because if it is not, calling **stop** generates an error.

The **>>>** button has callback **nextTrack**. **nextTrack** (lines 159–164) skips to the next track on the CD. If **CD** is initialized and the current track is not the last one, method **playTrack** is invoked, with the next track number specified (**currentTrack** + 1).

Similarly, the **<<<** button has callback **previousTrack**. **previousTrack** (lines 166–170) skips to the previous track on a CD. If **CD** is initialized and the current track is

not the first one, method **playTrack** is invoked, with the previous track number specified (**currentTrack** - 1).

Method **playTrack** (lines 144–157) plays a specified track of the CD. If the CD is initialized, line 148 sets **currentTrack** to the specified track number. Lines 149–150 then set **trackLabel** to the new track number and select the specified track number from the dropdown box. If the CD is currently playing another track, line 154 simply plays the specified track instead. If the CD is paused, however, lines 156–157 begin play of the specified track and then call method **playCD** to re-pause the disc.

The dropdown box (**trackChoices**) has callback method **changeTrack**. When the user selects a track number from the listbox, **changeTrack** (lines 172–178) is invoked. If **CD** is initialized, lines 176–177 obtain the index of the selection with **Tkinter ListBox** method **curselection**. Line 178 invokes method **playTrack** to play the selected track (**index** + 1).

The **Eject** button has callback method **ejectCD** (lines 180–193). Line 183 displays a **tkMessageBox** window which asks the user if the CD should be ejected. This is a safeguard against accidental ejection. If the user chooses to eject the CD, **CD** is initialized (the CD may not be playing), the disc is ejected with CD method **eject** and **CD** is uninitialized (lines 186–188). Lines 189–193 sets the CD player interface to its initial appearance.

The CD player updates its display with method **updateTime**, originally called in line 25. **updateTime** (lines 195–230) updates the CD player display (lines 198–227) and invokes common widget method **after**. **after** registers a callback that is called after a specified amount of milliseconds. Line 230 ensures that method **updateTime** is called every 1000 milliseconds (one second). Line 198 checks if **CD** is initialized. If not, execution skips to line 230.

Otherwise, the current number of seconds into the currently playing track is obtained from **CD** method **get_current** and stored in variable **seconds** (line 199). **get_current** returns a two-element tuple of the current track number and the number of seconds into that track. Lines 200–201 obtain the track length from **CD** method **get_track_length**, specifying the current track (**currentTrack** - 1). This value is stored in variable **endSeconds**. Lines 204–205 ensure that one track plays consecutively after another until the entire disk has been played. Lines 207–210 use **seconds** and **endSeconds** to determine the current time and end time in minutes and seconds.

Lines 213–214 create a string for the current track time (**trackTime**). The string has the form *mm:ss* where *mm* is minutes and *ss* is seconds. Note that **string** function **zfill** pads the string with zeros so that it occupies the correct number of spaces. This ensures that minutes or seconds in the range 0–9 (inclusive) result in strings of the same length as other minute or second values.

Line 218 determines if the CD is paused. If not, **timeDisplay** is updated to display the current time (line 227). Otherwise, **timeDisplay** is updated to either the current time or a symbol representing pause (lines 221–224). This ensures that the display flashes between the track time and the pause symbol when paused.

When finished using the CD player, the user destroys the window, invoking the **CDPlayer**'s **destroy** method (lines 27–34). Line 30 checks if **CD** is initialized. If so, **CD** method **stop** is invoked to stop the CD. If this was not done, the CD would continue to play after the user destroyed the window. Lines 33–34 uninitialize the **pygame cdrom** module and destroys the frame with Frame method **destroy**.

Look-and-Feel Observation 24.1

For Tkinter programs, a **`destroy`** *method acts as a destructor.*

24.8 `Pygame` Space Cruiser

This section demonstrates the most popular use of **`pygame`**, a two-dimensional game. Figure 24.6 uses various **`pygame`** modules to create a simple "Space Cruiser" game. In this game, the player controls a space ship flying through an asteroid field. The player has 60 seconds to fly through the asteroid field. After 60 seconds, the ship's fuel is exhausted, and the game is over. A clock in the upper-left corner of the screen shows the remaining time. Whenever the ship collides with an asteroid, 5 seconds are deducted from the time remaining. However, the ship may also pick up energy packs, which add 5 extra seconds to the timer. The player controls the ship with the arrow keys.

```
#!c:\Python\python.exe
# SpaceCruiser.py: Space Cruiser game using pygame

import os
import sys
import random
import pygame, pygame.image, pygame.font, pygame.mixer
from pygame.locals import *

class Sprite:
   """An object to place on the screen"""

   def __init__( self, image ):
      """Initialize object image and calculate rectangle"""

      self.image = image
      self.rectangle = image.get_rect()

   def place( self, screen ):
      """Place the object on the screen"""

      return screen.blit( self.image, self.rectangle )

   def remove( self, screen, background ):
      """Place the background over the image to remove it"""

      return screen.blit( background, self.rectangle,
         self.rectangle )

class Player( Sprite ):
   """A Player Sprite with 4 different states"""

   def __init__( self, images, crashImage,
      centerX = 0, centerY = 0 ):
      """Store all images and set the initial Player state"""
```

Fig. 24.6 **`Pygame`** example (part 1 of 9).

```

      self.movingImages = images
      self.crashImage = crashImage
      self.centerX = centerX
      self.centerY = centerY
      self.playerPosition = 1          # start player facing down
      self.speed = 0
      self.loadImage()

   def loadImage( self ):
      """Load Player image and calculate rectangle"""

      if self.playerPosition == -1:   # player has crashed
         image = self.crashImage
      else:
         image = self.movingImages[ self.playerPosition ]

      Sprite.__init__( self, image )
      self.rectangle.centerx = self.centerX
      self.rectangle.centery = self.centerY

   def moveLeft( self ):
      """Change Player image to face one position to the left"""

      if self.playerPosition == -1:   # player has crashed
         self.speed = 1
         self.playerPosition = 0      # move left of obstacle
      elif self.playerPosition > 0:
         self.playerPosition -= 1

      self.loadImage()

   def moveRight( self ):
      """Change Player image to face one position to the right"""

      if self.playerPosition == -1:   # player has crashed
         self.speed = 1
         self.playerPosition = 2      # move right of obstacle
      elif self.playerPosition < ( len( self.movingImages ) - 1 ):
         self.playerPosition += 1

      self.loadImage()

   def decreaseSpeed( self ):

      if self.speed > 0:
         self.speed -= 1

   def increaseSpeed( self ):

      if self.speed < 10:
         self.speed += 1

      # player has crashed, start player facing down
```

Fig. 24.6 **Pygame** example (part 2 of 9).

```
      if self.playerPosition == -1:
         self.playerPosition = 1
         self.loadImage()

   def collision( self ):
      """Change Player image to crashed player"""

      self.speed = 0
      self.playerPosition = -1
      self.loadImage()

   def collisionBox( self ):
      """Return smaller bounding box for collision tests"""

      return self.rectangle.inflate( -20, -20 )

   def isMoving( self ):
      """Player is not moving if speed is 0"""

      if self.speed == 0:
         return 0
      else:
         return 1

   def distanceMoved( self ):
      """Player moves twice as fast when facing straight down"""

      xIncrement, yIncrement = 0, 0

      if self.isMoving():

         if self.playerPosition == 1:
            xIncrement = 0
            yIncrement = 2 * self.speed
         else:
            xIncrement = ( self.playerPosition - 1 ) * self.speed
            yIncrement = self.speed

      return xIncrement, yIncrement

class Obstacle( Sprite ):
   """A moveable Obstacle Sprite"""

   def __init__( self, image, centerX = 0, centerY = 0 ):
      """Load Obstacle image and initialize rectangle"""

      Sprite.__init__( self, image )

      # move Obstacle to specified location
      self.positiveRectangle = self.rectangle
      self.positiveRectangle.centerx = centerX
      self.positiveRectangle.centery = centerY

      # display Obstacle in moved position to buffer visible area
```

Fig. 24.6 **Pygame** example (part 3 of 9).

```
      self.rectangle = self.positiveRectangle.move( -60, -60 )

   def move( self, xIncrement, yIncrement ):
      """Move Obstacle location up by specified increments"""

      self.positiveRectangle.centerx -= xIncrement
      self.positiveRectangle.centery -= yIncrement

      # change position for next pass
      if self.positiveRectangle.centery < 25:
         self.positiveRectangle[ 0 ] += \
            random.randrange( -640, 640 )

      # keep rectangle values from overflowing
      self.positiveRectangle[ 0 ] %= 760
      self.positiveRectangle[ 1 ] %= 600

      # display obstacle in moved position to buffer visible area
      self.rectangle = self.positiveRectangle.move( -60, -60 )

   def collisionBox( self ):
      """Return smaller bounding box for collision tests"""

      return self.rectangle.inflate( -20, -20 )

class Objective( Sprite ):
   """A moveable Objective Sprite"""

   def __init__( self, image, centerX = 0, centerY = 0 ):
      """Load Objective image and initialize rectangle"""

      Sprite.__init__( self, image )

      # move Objective to specified location
      self.rectangle.centerx = centerX
      self.rectangle.centery = centerY

   def move( self, xIncrement, yIncrement ):
      """Move Objective location up by specified increments"""

      self.rectangle.centerx -= xIncrement
      self.rectangle.centery -= yIncrement

# place a message on screen
def displayMessage( message, screen, background ):
   font = pygame.font.Font( None, 48 )
   text = font.render( message, 1, ( 250, 250, 250 ) )
   textPosition = text.get_rect()
   textPosition.centerx = background.get_rect().centerx
   textPosition.centery = background.get_rect().centery
   return screen.blit( text, textPosition )

# remove old time and place updated time on screen
def updateClock( time, screen, background, oldPosition ):
```

Fig. 24.6 **Pygame** example (part 4 of 9).

```
    remove = screen.blit( background, oldPosition, oldPosition )
    font = pygame.font.Font( None, 48 )
    text = font.render( str( time ), 1, ( 250, 250, 250 ),
       ( 0, 0, 0 ) )
    textPosition = text.get_rect()
    post = screen.blit( text, textPosition )
    return remove, post

def main():

    # constants
    WAIT_TIME = 20              # time to wait between frames
    COURSE_DEPTH = 50 * 480     # 50 screens long
    NUMBER_ASTEROIDS = 20       # controls number of asteroids

    # variables
    distanceTraveled = 0        # vertical distance
    nextTime = 0                # time to generate next frame
    courseOver = 0              # the course has not been completed
    allAsteroids = []           # randomly generated obstacles
    dirtyRectangles = []        # screen positions that have changed
    energyPack = None           # current energy pack on screen
    timeLeft = 60               # time left to finish course
    newClock = ( 0, 0, 0, 0 )   # the location of the clock

    # find path to all sounds
    collisionFile = os.path.join( "data", "collision.wav" )
    chimeFile = os.path.join( "data", "energy.wav" )
    startFile = os.path.join( "data", "toneup.wav" )
    applauseFile = os.path.join( "data", "applause.wav" )
    gameOverFile = os.path.join( "data", "tonedown.wav" )

    # find path to all images
    shipFiles = []
    shipFiles.append( os.path.join( "data", "shipLeft.gif" ) )
    shipFiles.append( os.path.join( "data", "shipDown.gif" ) )
    shipFiles.append( os.path.join( "data", "shipRight.gif" ) )
    shipCrashFile = os.path.join( "data", "shipCrashed.gif" )
    asteroidFile = os.path.join( "data", "Asteroid.gif" )
    energyPackFile = os.path.join( "data", "Energy.gif" )

    # obtain user preference
    fullScreen = int( raw_input(
       "Fullscreen? ( 0 = no, 1 = yes ): " ) )

    # initialize pygame
    pygame.init()

    if fullScreen:
       screen = pygame.display.set_mode( ( 640, 480 ), FULLSCREEN )
    else:
       screen = pygame.display.set_mode( ( 640, 480 ) )

    pygame.display.set_caption( "Space Cruiser!" )
```

Fig. 24.6 **Pygame** example (part 5 of 9).

```
   pygame.mouse.set_visible( 0 )   # make mouse invisible

   # create background and fill with black
   background = pygame.Surface( screen.get_size() ).convert()
   background.fill( ( 0, 0, 0 ) )

   # blit background onto screen and update entire display
   screen.blit( background, ( 0, 0 ) )
   pygame.display.update()

   collisionSound = pygame.mixer.Sound( collisionFile )
   chimeSound = pygame.mixer.Sound( chimeFile )
   startSound = pygame.mixer.Sound( startFile )
   applauseSound = pygame.mixer.Sound( applauseFile )
   gameOverSound = pygame.mixer.Sound( gameOverFile )

   # load images, convert pixel format and make white transparent
   loadedImages = []

   for file in shipFiles:
      surface = pygame.image.load( file ).convert()
      surface.set_colorkey( surface.get_at( ( 0, 0 ) ) )
      loadedImages.append( surface )

   # load crash image
   shipCrashImage = pygame.image.load( shipCrashFile ).convert()
   shipCrashImage.set_colorkey( shipCrashImage.get_at( ( 0, 0 ) ) )

   # initialize theShip
   centerX = screen.get_width() / 2
   theShip = Player( loadedImages, shipCrashImage, centerX, 25 )

   # load asteroid image
   asteroidImage = pygame.image.load( asteroidFile ).convert()
   asteroidImage.set_colorkey( asteroidImage.get_at( ( 0, 0 ) ) )

   # place an asteroid in a randomly generated spot
   for i in range( NUMBER_ASTEROIDS ):
      allAsteroids.append( Obstacle( asteroidImage,
         random.randrange( 0, 760 ), random.randrange( 0, 600 ) ) )

   # load energyPack image
   energyPackImage = pygame.image.load( energyPackFile ).convert()
   energyPackImage.set_colorkey( surface.get_at( ( 0, 0 ) ) )

   startSound.play()
   pygame.time.set_timer( USEREVENT, 1000 )

   while not courseOver:

      # wait if moving too fast for selected frame rate
      currentTime = pygame.time.get_ticks()

      if currentTime < nextTime:
```

Fig. 24.6 **Pygame** example (part 6 of 9).

```
           pygame.time.delay( nextTime - currentTime )

        nextTime = currentTime + WAIT_TIME

        # remove all objects from the screen
        dirtyRectangles.append( theShip.remove( screen,
           background ) )

        for asteroid in allAsteroids:
           dirtyRectangles.append( asteroid.remove( screen,
              background ) )

        if energyPack is not None:
           dirtyRectangles.append( energyPack.remove( screen,
              background ) )

        # get next event from event queue
        event = pygame.event.poll()

        # if player has quit program or pressed escape key
        if event.type == QUIT or \
           ( event.type == KEYDOWN and event.key == K_ESCAPE ):
           sys.exit()

        # if up arrow key was pressed, slow ship
        elif event.type == KEYDOWN and event.key == K_UP:
           theShip.decreaseSpeed()

        # if down arrow key was pressed, speed up ship
        elif event.type == KEYDOWN and event.key == K_DOWN:
           theShip.increaseSpeed()

        # if right arrow key was pressed, move ship right
        elif event.type == KEYDOWN and event.key == K_RIGHT:
           theShip.moveRight()

        # if left arrow key was pressed, move ship left
        elif event.type == KEYDOWN and event.key == K_LEFT:
           theShip.moveLeft()

        # one second has passed
        elif event.type == USEREVENT:
           timeLeft -= 1

        # 1 in 100 odds of creating a new energyPack
        if energyPack is None and not random.randrange( 100 ):
           energyPack = Objective( energyPackImage,
              random.randrange( 0, 640 ), 480 )

        # update obstacle and energyPack positions if ship is moving
        if theShip.isMoving():
           xIncrement, yIncrement = theShip.distanceMoved()

           for asteroid in allAsteroids:
```

Fig. 24.6 **Pygame** example (part 7 of 9).

```
            asteroid.move( xIncrement, yIncrement )

         if energyPack is not None:
            energyPack.move( xIncrement, yIncrement )

            if energyPack.rectangle.bottom < 0:
               energyPack = None

         distanceTraveled += yIncrement

      # check for collisions with smaller bounding boxes
      # for better playability
      asteroidBoxes = []

      for asteroid in allAsteroids:
         asteroidBoxes.append( asteroid.collisionBox() )

      # retrieve list of all obstacles colliding with player
      collision = theShip.collisionBox().collidelist(
         asteroidBoxes )

      # move asteroid one screen down
      if collision != -1:
         collisionSound.play()
         allAsteroids[ collision ].move( 0, -540 )
         theShip.collision()
         timeLeft -= 5

      # check if player has gotten energyPack
      if energyPack is not None:

         if theShip.collisionBox().colliderect(
            energyPack.rectangle ):
            chimeSound.play()
            energyPack = None
            timeLeft += 5

      # place all objects on screen
      dirtyRectangles.append( theShip.place( screen ) )

      for asteroid in allAsteroids:
         dirtyRectangles.append( asteroid.place( screen ) )

      if energyPack is not None:
         dirtyRectangles.append( energyPack.place( screen ) )

      # update time
      oldClock, newClock = updateClock( timeLeft, screen,
         background, newClock )
      dirtyRectangles.append( oldClock )
      dirtyRectangles.append( newClock )

      # update changed areas of display
      pygame.display.update( dirtyRectangles )
```

Fig. 24.6 **Pygame** example (part 8 of 9).

```
414          dirtyRectangles = []
415
416          # check for course end
417          if distanceTraveled > COURSE_DEPTH:
418             courseOver = 1
419
420          # check for game over
421          elif timeLeft <= 0:
422             break
423
424       if courseOver:
425          applauseSound.play()
426          message = "Asteroid Field Crossed!"
427       else:
428          gameOverSound.play()
429          message = "Game Over!"
430
431       pygame.display.update( displayMessage( message, screen,
432          background ) )
433
434       # wait until player wants to close program
435       while 1:
436          event = pygame.event.poll()
437
438          if event.type == QUIT or \
439             ( event.type == KEYDOWN and event.key == K_ESCAPE ):
440             break
441
442    if __name__ == "__main__":
443       main()
```

Fig. 24.6 **Pygame** example (part 9 of 9).

When the program is run, function **main** (lines 206–440) is executed. Lines 209–221 create some constants and variables which will be used (and explained later). Lines 224–237 locate the sound and image files, which are located in the **"data"** subdirectory. **os.path.join** ensures that the path will be correct on any platform. The program prompts the player to select fullscreen or windowed mode. The player's response is stored in variable **fullScreen**.

Line 244 initializes **pygame**. This call to **init** is a shortcut for calling each module's **init** function separately. Lines 246–249 set the current display mode with **pygame.display** function **set_mode**. The first argument passed to **set_mode** is a two-element tuple specifying a display mode 640 pixels wide and 480 pixels high. If the player has selected fullscreen mode, the program passes **set_mode** flag **FULLSCREEN**, an SDL constant. Otherwise, no flags are passed. The value returned by **set_mode** is a pygame **Surface** object, a blank canvas onto which the game is drawn. This **Surface** object is stored in variable **screen**. Line 251 sets the window caption to read **"Space Cruiser!"** by invoking **pygame.display** function **set_caption**. Line 252 calls **pygame.mouse** function **set_visible** with argument 0, ensuring that the mouse cursor will not appear over the window.

Lines 255–256 create the black background for the game. First, the program creates a **pygame Surface** that is the same size as the window. The size of the window is obtained from **screen** method **get_size**. **Surface** method **convert** is then invoked on the background. **convert** is used to convert a surface's pixel format to the display format so that *blits* are performed faster. Blits will be discussed later. The call to **background**'s **fill** method fills the background with the color black. The argument passed to **fill** is a three-element tuple representing the RGB values of the desired color. Because black has no red, green or blue, it is represented by **(0, 0, 0)**.

Line 259 *blits* the background onto the screen. Blitting can be thought of as drawing an object on a surface. The call to **screen**'s **blit** method in line 259 draws the background onto the screen at position **(0, 0)**. Position **(0, 0)** represents the upper-left corner of the screen. Because the background is the same size as the screen, the background will fill the screen. However, if the screen were visible at this point, it would not yet be black. Although the background has been blitted, the display has not been updated. This is done in line 260. The **pygame.display** function **update** updates the display. If passed no arguments, **update** will update the entire display **Surface**. We will see later that this is not always necessary (or efficient).

Lines 262–266 load all necessary sound files. Each line creates a **Sound** object (defined in **pygame.mixer**) from a path created in lines 224–228. Lines 269–278 load the ship images. In our game, the ship has four possible states: *moving left*, *moving down*, *moving right* and *crashed*. Because of the implementation of class **Player** (discussed later), the paths to the images representing the first three states are appended to list **shipFiles** (lines 232–234). The **for/in** loop at line 271–274 iterates over this list, loading each image. Line 272 loads an image with **pygame.image** function **load**. Note that just as the background's pixel format was converted, the pixel format of each image loaded must be converted. The value returned by **load** is a pygame **Surface**, which is stored in variable **surface**. Line 273 invokes **surface** method **get_at** to obtain the color of the image at position **(0, 0)**. For each image, the color at this position is white. **surface** method **set_colorkey** is then passed this color. The effect is that the color white will appear transparent for each surface. Each surface is appended to list **loadedImages**. Lines 277–278 similarly load the image representing the *crashed* state.

Line 281 invokes **screen** method **get_width** to obtain the width of the window. Because we want our ship to appear halfway across the screen, **centerX** is assigned half of this value. Line 282 creates a **Player** object and assigns it to variable **theShip**. The arguments passed to the **Player** constructor ensures that the ship appears halfway across the screen, 25 pixels from the top. We will now discuss two classes, **Sprite** and **Player**.

Class **Sprite** (lines 10–28) defines any object that we place on the screen. The **Sprite** constructor takes as input a pygame **Surface** called **image**. Lines 16 stores this **Surface** in class attribute **image**. Line 17 computes the image's *bounding rectangle* with **Surface** method **get_rect**, and stores it in attribute **rectangle**. The object returned by **get_rect** is a **pygame** *rectstyle*.

A **pygame** *rectstyle* represents a rectangular area and may have three possible forms. The first is a four-element sequence of the form [*xpos*, *ypos*, *width*, *height*], where *xpos* and *ypos* are the coordinates of the upper-left corner of the rectangle, and *width* and *height* are the dimensions of the rectangle. The second is a pair of sequences of the form [[*xpos*, *ypos*], [*width*, *height*]]. The third is an instance of class **pygame.Rect**. A **Rect** object

represents a rectangle as well, but also has several useful methods. The *rectstyle* returned by **get_rect** is a **Rect** object with *xpos* and *ypos* of **0**. Many pygame functions accept *rectstyles* as arguments rather than just **Rect** objects (including the **Rect** constructor). In this case, it is possible (and more convenient) to simply pass the function a four-element sequence.

Sprite method **place** (lines 19–22) "places" the object on the screen. **place** takes as an argument **Surface screen**. **screen**'s **blit** method is invoked (line 22) to draw the object at position **rectangle**. Note that changes to **rectangle** will change where the object is drawn. **place** then returns the value returned by **blit**, a **Rect** representing the area blitted.

Sprite method **remove** (lines 24–28) "removes" an object from the screen by drawing the background over it (lines 27–28). Note that this call to **blit** has three arguments, two of which are **rectangle**. The third argument specifies what section of **background** to draw at position **rectangle**. If no third argument were specified, the entire background would be drawn at **rectangle**. **remove** returns a **Rect** representing the area blitted.

Class **Player** (lines 30–128) represents the object controlled by the player which appears to move across the screen. In the game, this object is a spaceship. **Player** inherits from class **Sprite**. Line 282 creates a **Player** object, invoking **Player**'s constructor (lines 33–43). Lines 37–40 store the image surfaces and starting position into class attributes. Line 41 sets **playerPosition** to 1. **playerPosition** is the index of the current image being displayed. Because **movingImages** is a list of length 3, the indices 0, 1, 2 represent *moving left*, *moving down* and *moving right*, respectively. Thus, line 41 starts the **Player** in state *moving down*. **playerPosition** of –1 indicates the player has *crashed*. Line 42 sets attribute **speed** to 0, and line 43 calls method **loadImage**.

loadImage (lines 45–55) updates attributes of **Player**. Lines 48–51 determine the correct image to use. If the player has not *crashed*, the image representing the current player state is used (line 51). Line 53 invokes **Sprite**'s constructor to update the **image** and **rectangle** attributes. Lines 54–55 move the object to the correct position by changing **rectangle**'s **centerx** and **centery** attributes.

Player methods **moveLeft** and **moveRight** are called when the player presses the left and right arrow keys, respectively. Because they are similar, we will discuss them together. First, an **if** statement checks if the player has *crashed* (i.e., **playerPosition** is –1). If so, **speed** is set to 1 and move the player either to the left (line 62) or right (line 73) of the obstacle. Otherwise, if the player is not as far left or right as possible, we move the player left (line 64) or right (line 75) one position. Finally, method **loadImage** updates the image.

Method **decreaseSpeed** (lines 79–82) is called when the user presses the up arrow key. **decreaseSpeed** decreases attribute **speed** by 1. Pressing the down arrow key invokes method **increaseSpeed** (lines 84–92). **increaseSpeed** increases **speed** by 1. Lines 90–92 test if the player has *crashed*. If so, **playerPosition** is set to 1 (*moving down*) and the image is updated (line 92).

Player method **collision** (lines 94–99) is called when the ship collides with an asteroid. **collision** sets **speed** to 0, sets **playerPosition** to –1 (*crashed*) and invokes method **loadImage**. Collisions are tested for with the **Rect** returned by method **collisionBox** (lines 101–104). **collisionBox** calls **Rect** method **inflate** and

returns the results. **inflate** returns a new **Rect** which represents the calling **Rect** reduced or enlarged around its center by a specified amount. Note that we test for collisions with smaller bounding rectangles for playability purposes. Most likely, the image of the player are using does not completely fill its rectangle. It would become frustrating to the player if collisions were to occur when bounding rectangles intersected, but images did not. Using smaller bounding rectangles for collision detection is sometimes referred to as *sub-rectangle collision*.

Method **distanceMoved** (lines 114–128) determines the current change in player position. Line 119 invokes method **isMoving** (lines 106–112) to test if the player is moving. If so, **xIncrement** and **yIncrement** must be calculated. Lines 121–126 use **playerPosition** and **speed** to determine the distance moved. Note that when *moving down*, the player moves twice as fast in the vertical direction as when *moving left* or *moving right*.

Once a **Player** is instantiated (line 282), the program creates the asteroids. Lines 285–286 load the asteroid image, setting white to transparent. The **for**/**in** loop in lines 289–291 creates **NUMBER_ASTEROID** asteroids. Each asteroid is an instance of class **Obstacle** (discussed later). The arguments passed to **Obstacle**'s constructor ensure that each asteroid will be randomly placed on the screen. Note that the values passed to **random.randrange** are larger than the screen size in order to buffer the visible area. The game will simulate ship movement by moving these asteroids up the screen. The direction the asteroids move depends upon the current state of the ship. When an asteroid moves off the top of the screen, it will be placed on the bottom of the screen again, creating a scrolling effect.

Class **Obstacle** (lines 130–167) inherits from **Sprite**. An **Obstacle** represents an object which the player must avoid. In our game, this object is an asteroid. When an **Obstacle** is created, its constructor (lines 133–144) is invoked. Line 136 calls the **Sprite** constructor to initialize the **image** and **rectangle** attributes.

Because we want asteroids to move off the screen completely (i.e., into negative screen coordinates) before removing them and placing them back on the screen, we must buffer the visible area. In order to do so, we must keep track of two locations for each **Obstacle**. **rectangle** represents the actual location of the asteroid. This is where we **place** the object. **positiveRectangle** represents the coordinates of **rectangle** shifted into positive screen coordinates. Lines 139–141 create and initialize the position of **positiveRectangle**. Line 144 updates **rectangle** by invoking **Rect** method **move**. This effect is that **rectangle** is now a rectangle of the same dimensions as **positiveRectangle**, but shifted by –60 pixels in both the x and y directions.

Obstacle method **move** (lines 146–162) is used to move the object. **move** requires arguments **xIncrement** and **yIncrement**. Recall that class **Player** has method **distanceMoved**. This method returns the necessary values. Lines 149–150 move the position of **positiveRectangle** up the screen by the specified amounts. The **if** statement at line 153 checks if the asteroid has reached the top of the screen. If so, lines 154–155 add a random integer to the *xpos* of **positiveRectangle**. This ensures that the next time the asteroid appears on the screen, it will not have the same x coordinate as its previous pass. If these lines were omitted, the asteroid positions would appear to loop, making gameplay boring. Notice that the program treats **positiveRectangle**, a **Rect** object, as if it were a four-element sequence of the form [*xpos*, *ypos*, *width*, *height*]. Lines

158–159 make sure that the *xpos* and *ypos* of **positiveRectangle** are within range. Finally, now that **positiveRectangle** has been updated, **Rect** method **move** (line 162) obtains the new **rectangle** value.

As with class **Player**, **Obstacle** collisions are tested for with the **Rect** returned by method **collisionBox** (lines 164–167). **collisionBox** calls **Rect** method **inflate** and returns the results.

After the creation of the asteroids (lines 289–291), methods **load** and **convert** load and convert the energy pack image (line 294), setting white to transparent (line 295). During gameplay, energy packs will be created from class **Objective**. **Objective** (lines 169–184) has a constructor (lines 173–179) and method **move** (lines 181–185) similar to those of class **Obstacle**. Line 297 invokes **Sound** method **play** to play **startSound**. The player will hear this sound when the game begins. Line 298 invokes **pygame.time** function **set_timer** to generate a **USEREVENT** event every 1000ms (one second). **USEREVENT** is a **pygame** constant which represents a user-defined event. The effect of line 298 is that every second, a **USEREVENT** event will be placed onto SDL's *event queue*. **pygame**'s *event system* will be discussed in detail later.

The **while** loop in lines 300–422 plays the game. Each iteration checks that **courseOver** is still 0. If it is, the asteroid field has not yet been crossed, and gameplay continues. Lines 303–308 use **pygame** module **time** to ensure that the game does not run too fast. Line 302 invokes **pygame.time** function **get_ticks**. **get_ticks** returns the time, in milliseconds, since **pygame.time** was imported. This value is stored in variable **currentTime**. If **currentTime** is less than **nextTime**, the previous number of "ticks" plus a constant (**WAIT_TIME**), we invoke **time** function **delay** (line 306). **delay** pauses the execution for a given number of milliseconds. The value passed to **delay** is the number of milliseconds remaining until **nextTime**.

Next, the program updates the display. In order to update the positions of all objects on the screen, it would be possible to **remove** each object, change its position and **place** (i.e., blit) it on the screen again. Then **pygame.display.update** (as in line 260) could update the entire display. However, updating the entire display is inefficient and slow. A popular method used to speed up screen updates is called *dirty rectangle animation*. In dirty rectangle animation, we maintain a list of rectangles (representing areas of the display) which have been altered (i.e., have become "dirty"). After removing an object from the screen, its current rectangle is appended to the list and the object's position is updated. Finally, the program places the object back on the screen and appends its new rectangle to the list. Method **update** is called with the list of "dirty" rectangles. The effect is that **update** will only update those parts of the display which have changed, dramatically improving game performance. Note that the list of rectangles passed to update can be a list of any *rectstyle*.

The game implements dirty rectangle animation. Lines 311–320 remove the ship, each asteroid and the energy pack (if one is present) from the screen by invoking their **remove** methods. Each time, **remove** returns a **Rect** representing the area changed. Each **Rect** is appended to list **dirtyRectangles**.

We now discuss pygame event handling. As with **Tkinter**, events can be generated from the keyboard or mouse. **pygame** also handles various other events, including joystick events. One method of **pygame** event handling uses the SDL event queue. As events are detected, they are placed on the queue. Each **Event** object on the queue has a **type**

attribute. Keypress **Event**s have **type KEYDOWN**. Most user-defined events have type **USEREVENT**. A request to quit the game results in a **QUIT** event.

Line 323 invokes **pygame.event** function **poll**. **poll** returns the next **Event** waiting on the queue. This object is stored in variable **event**. If **event** is a request to quit the game (**QUIT**) or a **KEYDOWN** event with **key** attribute **K_ESCAPE**, the program exits (line 328). Lines 330–344 check if **event** was generated by any of the four arrow keys (**K_UP**, **K_DOWN**, **K_RIGHT** or **K_LEFT**). If so, the corresponding **Player** method is invoked. Recall now that line 298 causes one **USEREVENT** event to be placed on the event queue every second. Line 347 checks if **event** is one of these. If so, **timeLeft**, the time remaining to cross the asteroid field, is reduced by 1 (line 348).

Lines 351–353 attempt to create a new energy pack. If an energy pack does not exists (**energyPack** is **None**) and **randrange** returns 0, the program creates a new **energyPack** from class **Objective**. The arguments passed to **Objective**'s constructor ensure that the pack will start at a random position at the bottom of the screen. Note that because the function call passes 100 to **randrange**, the odds of creating a new pack if one does not exist is 1 in 100.

We then update the positions of the asteroids and energy pack (if one exists). If the ship is moving (i.e., **speed** > 0), we retrieve the **xIncrement** and **yIncrement** from **Player** method **distanceMoved** (line 357). We update the position of each asteroid (lines 358–359) and the position of the energy pack (lines 362–363). Line 366 checks if the energy pack has moved off the top of the screen. If so, we destroy the current energy pack (line 366). Line 368 increments **distanceTraveled**.

The next section tests for asteroid collisions. Lines 372–375 create a list, **asteroidBoxes**, of **Rect**s returned from each asteroid's **collisionBox** method. The call then passes this list to **Rect** method **collidelist**. **collidelist** returns the index of the first *rectstyle* in a list which overlaps the base rectangle. In line 378–3799, the base rectangle is the **Rect** returned from the ship's **collisionBox** method. When an overlap is found, **collideList** stops checking the remaining list. If no overlap is found, **collideList** returns -1. If the ship has collided with an asteroid (line 383), we play a collision sound (line 383) and move the offending asteroid out of the way (line 384). Lines 384 and 385 invoke the ship's **collision** method and deduct 5 extra seconds from the time remaining.

Lines 389–395 check if the player has gotten an energy pack. Line 391–392 invokes **Rect** method **colliderect**. **colliderect** returns true if the calling **Rect** overlaps the argument *rectstyle*. If the player has, indeed, gotten the energy pack, the game plays **chimeSound**, removes the energy pack and adds 5 seconds to the clock (lines 393–395).

Lines 398–404 **place** all the objects back on the screen, appending their rectangles to **dirtyRectangles**. Lines 407–410 update the clock in the upper-left corner of the screen. Function **updateClock** (lines 196–203) removes the previous clock **Surface**, creates a new one and blits it onto the screen. A **pygame.font.Font** object (line 198) allows the program render text into a **Surface**. The **Font** constructor takes two arguments. The first is the name of the font file to use. If **None** is specified, **Font** will use the pygame default font file (bluebold.ttf). The second argument is the size of the font. Line 198 creates a **Font** of type *bluebold* and size 48. Lines 199–200 invoke **font**'s **render** method to create a new **Surface** with specified text. **render** accepts up to four arguments. The first is the text to create. The second specifies to use *antialiasing* (edge

smoothing) or not. The third is the RGB color to render the font in. The fourth is the RGB color of the background. If no fourth argument is specified, the text background will be transparent. **updateClock** returns both the old (**remove**) and new (**post**) rectangles.

Once the clock has been created and blitted on the screen, lines 409–410 append the clock's previous rectangle and current rectangle to **dirtyRectangles**. Line 413 is the final step in dirty rectangle animation. Every altered area of the display is updated. Without this line, the player would not see any change in the display. Line 414 re-initializes **dirtyRectangles** for the next iteration.

If the player has crossed the asteroid field (line 417), the program sets **courseOver** to 1. This will ensure the **while** loop exists after the current iteration. If not, the program checks whether the player has run out of time (line 421). If so, the program exits the **while** loop.

Once the **while** loop has been broken, execution continues at line 424 and checks if the player has won or lost the game. If the player has won, the game plays **applauseSound** and sets **message** to **"Asteroid Field Crossed!"**. Otherwise, the program plays **gameOverSound** and sets **message** to **"Game Over!"**. Lines 431–432 invoke **pygame.display** function **update** to display **message** to the player. Function **displayMessage** returns the *rectstyle* passed to **update**. **displayMessage** (lines 187–193) blits a message on the screen and returns the area of the screen which has been modified. **displayMessage** is similar to **updateClock**. The **while** loop in lines 435–440 waits for the user to exit the program.

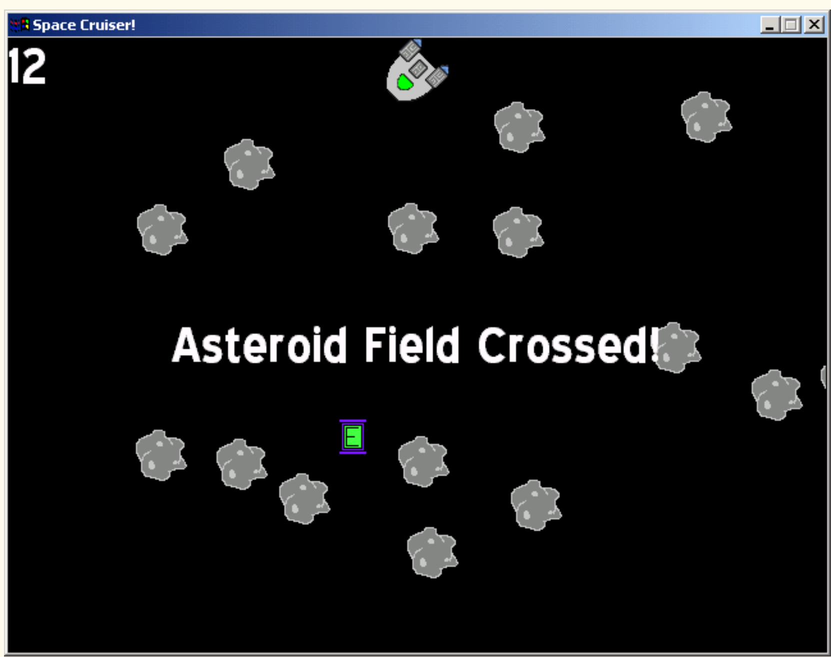


Fig. 24.7 Screenshot of Space Cruiser game.

24.9 Internet and World Wide Web Resources

pyopengl.sourceforge.net
The main Web site of the **PyOpenGL** module describes the module and provides links to documentation and the download page.

www.python.de/pyopengl.html
The old **PyOpenGL** module Web site contains information about the earlier versions of the module and some examples.

www.wag.caltech.edu/home/rpm/python_course/Lecture_7.pdf
This series of lecture slides discussing the interaction between Python and OpenGL. The slides include a few introductory examples.

www.opengl.org
The OpenGL home page includes a FAQ, downloads, documentation and forums.

[***Notes To Reviewers***]

- We will post this chapter for second-round review with back matter—summary, terminology, exercises and solutions.
- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send us e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **cheryl.yaeger@deitel.net**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
- Please be constructive. This book will be published soon. We all want to publish the best possible book.
- If you find something that is incorrect, please show us how to correct it.
- Please read all the back matter including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

O

P

Q

R

S

T

U

W

Z

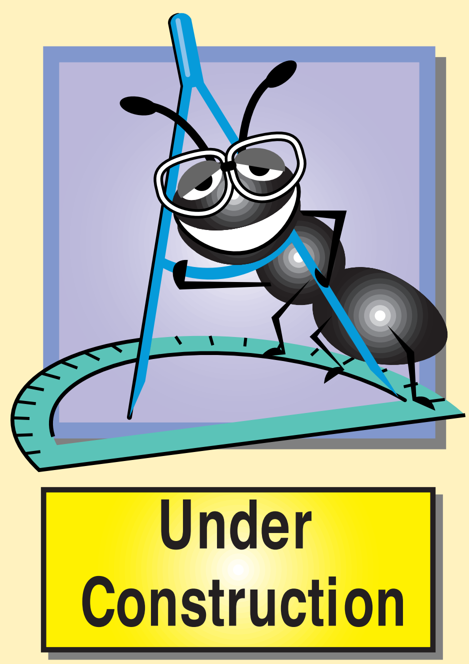
25

Accessibility

Objectives

- To introduce the World Wide Web Consortium's Web Content Accessibility Guidelines 1.0 (WCAG 1.0).
- To understand how to use the **`alt`** attribute of the **`<img>`** tag to describe images to people with visual impairments, mobile-Web-device users, search engines, etc.
- To understand how to make XHTML tables more accessible to page readers.
- To understand how to verify that XHTML tags are used properly and to ensure that Web pages are viewable on any type of display or reader.
- To understand how VoiceXML™ and CallXML™ are changing the way people with disabilities access information on the Web.
- To introduce the various accessibility aids offered in Windows 2000.

'Tis the good reader that makes the good book...
Ralph Waldo Emerson



Outline

25.1 Introduction

Enabling a Web site to meet the needs of individuals with disabilities is a concern for all businesses. People with disabilities are a significant portion of the population, and legal ramifications exist for Web sites that discriminate by not providing adequate and universal access to their resources. In this chapter, we explore the *Web Accessibility Initiative*, its guidelines, various laws regarding businesses and their availability to people with disabilities and how some companies have developed systems, products and services to meet the needs of this demographic.

25.2 Web Accessibility

In 1999, the National Federation for the Blind (NFB) filed a lawsuit against *America On Line (AOL)* for not supplying access to its services for people with visual disabilities. The *Americans with Disabilities Act (ADA)* and many other efforts address Web accessibility laws (Fig. 25.1).

Act	Purpose
Americans with Disabilities Act	The ADA prohibits discrimination on the basis of disability in employment, state and local government, public accommodations, commercial facilities, transportation and telecommunications.
Telecommunications Act of 1996	The Telecommunications Act of 1996 contains two amendments to Section 255 and Section 251(a)(2) of the Communications Act of 1934. These amendments require that communication devices, such as cell phones, telephones and pagers, be accessible to individuals with disabilities.
Individuals with Disabilities Education Act of 1997	Education materials in schools must be made accessible to children with disabilities.
Rehabilitation Act	Section 504 of the Rehabilitation Act states that college sponsored activities receiving federal funding cannot discriminate against individuals with disabilities. Section 508 mandates that all government institutions receiving federal funding design their Web sites such that they are accessible to individuals with disabilities. Businesses that service the government also must abide by this act.

Fig. 25.1 Acts designed to protect access to the Internet for people with disabilities.

WeMedia.com™ (Fig. 25.2) is a Web site dedicated to providing news, information, products and services for the millions of people with disabilities, their families, friends and caregivers. There are 54 million Americans with disabilities, representing an estimated $1 trillion in purchasing power. *We Media* also provides online educational opportunities for people with disabilities.

The Internet enables individuals with disabilities to work in a vast array of new fields. Technologies such as voice activation, visual enhancers and auditory aids, afford more employment opportunities. People with visual impairments may use computer monitors with enlarged text, while people with physical impairments may use head pointers with on-screen keyboards.

Federal regulations, similar to the disability ramp mandate, will be applied to the Internet to accommodate the needs of people with hearing, vision and speech impairments. In the following sections, we explore a variety of products and services that provide Internet access for people with disabilities.

25.3 Web Accessibility Initiative

On April 7, 1997, the World Wide Web Consortium (W3C) launched the *Web Accessibility Initiative* (WAI™). *Accessibility* refers to the usability of an application or Web site by people with disabilities. The majority of Web sites are considered either partially or totally inaccessible to people with visual, learning or mobility impairments. Total accessibility is

difficult to achieve because people have varying types of disabilities, language barriers and hardware and software inconsistencies. However, a high level of accessibility is attainable. As more people with disabilities use the Internet, it is imperative that Web site designers increase the accessibility of their sites. The WAI aims for such accessibility, as discussed in its mission statement described at **`www.w3.org/WAI`**.

This chapter explains some of the techniques for developing accessible Web sites. The WAI published the *Web Content Accessibility Guidelines* (*WCAG*) *1.0* to help businesses determine if their Web sites are accessible to everyone. The WCAG 1.0 (**`www.w3.org/TR/WCAG10`**) uses checkpoints to indicate specific accessibility requirements. Each checkpoint has an associated priority indicating its importance. *Priority-one checkpoints* are goals that must be met to ensure accessibility; we focus on these points in this chapter. *Priority-two checkpoints*, though not essential, are highly recommended. These checkpoints must be satisfied, or people with certain disabilities will experience difficulty accessing Web sites. *Priority-three checkpoints* slightly improve accessibility.



Fig. 25.2 We Media home page.

At the time of this writing, the WAI is working on the *WCAG 2.0* draft. A single checkpoint in the WCAG 2.0 Working Draft may encompass several checkpoints from WCAG 1.0; WCAG 2.0 checkpoints will supersede those in WCAG 1.0. Also, the WCAG 2.0 supports a wider range of markup languages (i.e., XML, WML, etc.) and content types than its predecessor. To obtain more information about the WCAG 2.0 Working Draft, visit **www.w3.org/TR/WCAG20**.

The WAI also presents a supplemental checklist of *quick tips*, which reinforce ten important points for accessible Web site design. More information on the WAI Quick Tips resides at **www.w3.org/WAI/References/Quicktips**.

25.4 Providing Alternatives for Images

One important WAI requirement is to ensure that every image on a Web page is accompanied by a textual description that clearly defines the purpose of the image. To accomplish this task, include a text equivalent of each item by using the **alt** attribute of the **<img>** and **<input>** tags. A text equivalent for images defined using the **object** element is the text between the start and end **<object>** tag.

Web developers who do not use the **alt** attribute to provide text equivalents increase the difficulty people with visual impairments experience in navigating the Web. Specialized *user agent*s, such as *screen readers* (programs that allow users to hear text and text descriptions displayed on their screens) and *braille displays* (devices that receive data from screen-reading software and output the data as braille), allow people with visual impairments to access text-based information displayed on the screen. A user agent visually interprets Web-page source code and translates it into formatted text and images. Web browsers, such as Microsoft Internet Explorer and Netscape Communicator, and the screen readers mentioned throughout this chapter are examples of user agents.

Web pages that do not provide text equivalents for video and audio clips are difficult for people with visual and hearing impairments to access. Screen readers cannot read images, movies and other non-XHTML objects from these Web pages. Providing multimedia-based information in a variety of ways (i.e., using the **alt** attribute or providing in-line descriptions of images) helps maximize the content's accessibility.

Web designers should provide useful text equivalents in the **alt** attribute for use in non-visual user agents. For example, if the **alt** attribute describes a sales growth chart, the attribute should provide a brief summary of the data; it should not describe the data in the chart. Instead, a complete description of the chart's data should be included in the ***longdesc*** attribute, which is intended to augment the **alt** attribute's description. The **longdesc** attribute contains the URL that links to a Web page describing the image or multimedia content. Currently, most Web browsers do not support the **longdesc** attribute. An alternative for the **longdesc** attribute is *D-link*, which provides descriptive text about graphs and charts. More information on D-links can be obtained at the *CORDA Technologies* Web site (**www.corda.com**).

Using a screen reader for Web-site navigation can be time consuming and frustrating, as screen readers cannot interpret pictures and other graphical content. A link at the top of each Web page that provides direct access to the page's content allows users to bypass a long list of navigation links or other inaccessible elements. This jump can save time and eliminate frustration for individuals with visual impairments.

Emacspeak is a screen interface that allows greater Internet access to individuals with visual disabilities by translating text to voice data. The open source product also implements auditory icons that play various sounds. Emacspeak can be customized with Linux operating systems and provides support for the IBM *ViaVoice* speech engine. The Emacspeak Web site is located at **www.cs.cornell.edu/home/raman/emacspeak/emacspeak.html**.

In March 2001, We Media introduced the "WeMedia Browser," which allows people with poor vision and cognitive disabilities (e.g., dyslexia) to use the Internet more conveniently. The *WeMedia Browser i*mproves upon the traditional browser by providing oversized buttons and keystroke commands for navigation. The user can control the speed and volume at which the browser "reads" Web page text. The WeMedia Browser is available for free download at **www.wemedia.com**.

IBM Home Page Reader (HPR) is another browser that "reads" text selected by the user. The HPR uses the IBM ViaVoice technology to synthesize a voice. A trial version of HPR is available at **www-3.ibm.com/able/hpr.html**.

25.5 Maximizing Readability by Focusing on Structure

Many Web sites use tags for aesthetic purposes rather than for the appropriate purpose. For example, the **<h1>** heading tag often is used erroneously to make text large and bold rather than as a major section head for content. The desired visual effect may be achieved, but it creates a problem for screen readers. When the screen reader software encounters the **<h1>** tag, it verbally may inform the user that a new section has been reached when it is not the case, which may confuse users. Only use the **h1** in accordance with its XHTML specifications (e.g., as headings to introduce important sections of a document). Instead of using **h1** to make text large and bold, use CSS (discussed in Chapter 28, Cascading Style Sheets) or XSL (discussed in Chapter 15, Extensible Markup Language) to format and style the text. For further examples, refer to the WCAG 1.0 Web site at **www.w3.org/TR/WCAG10**. [*Note:* The **<strong>** tag also may be used to make text bold; however, screen readers emphasize bold text, which affects the inflection of what is spoken.]

Another accessibility issue is *readability*. When creating a Web page intended for the general public, it is important to consider the reading level (i.e., the comprehension and level of understanding) at which it is written. Web site designers can make their sites easier to read by using shorter words. Designers should also limit slang terms and other non-traditional language that may be problematic for users from other countries.

WCAG 1.0 suggests using a paragraph's first sentence to convey its subject. Stating the point of the paragraph in its first sentence makes its easier to find crucial information and allows readers to bypass unwanted material.

The *Gunning Fog Index*, a formula that produces a readability grade when applied to a text sample, evaluates a Web site's readability. More information about the Gunning Fog Index can be obtained at **www.trainingpost.org/3-2-inst.htm**.

25.6 Accessibility in XHTML Tables

Complex Web pages often contain tables for formatting content and presenting data. Many screen readers are incapable of translating tables correctly unless the tables are properly de-

signed. For example, the *CAST eReader*, a screen reader developed by the Center for Applied Special Technology (**www.cast.org**), starts at the top-left-hand cell and reads columns from top to bottom, left to right. This procedure is known as reading a table in a *linearized* manner. The CAST eReader reads the table in Fig. 25.3 as follows:

```
Price of Fruit Fruit Price Apple $0.25 Orange $0.50 Banana
$1.00 Pineapple $2.00
```

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 25.3: withoutheaders.html -->
<!-- Table without headers          -->

<html>
   <head>
      <title>XHTML Table Without Headers</title>

      <style type = "text/css">
         body { background-color: #ccffaa;
                text-align: center }
      </style>
   </head>

   <body>

      <p>Price of Fruit</p>

      <table border = "1" width = "50%">

         <tr>
            <td>Fruit</td>
            <td>Price</td>
         </tr>

         <tr>
            <td>Apple</td>
            <td>$0.25</td>
         </tr>

         <tr>
            <td>Orange</td>
            <td>$0.50</td>
         </tr>

         <tr>
            <td>Banana</td>
            <td>$1.00</td>
         </tr>

```

Fig. 25.3 XHTML table without accessibility modifications (part 1 of 2).

```
            <tr>
               <td>Pineapple</td>
               <td>$2.00</td>
            </tr>

         </table>

      </body>
   </html>
```

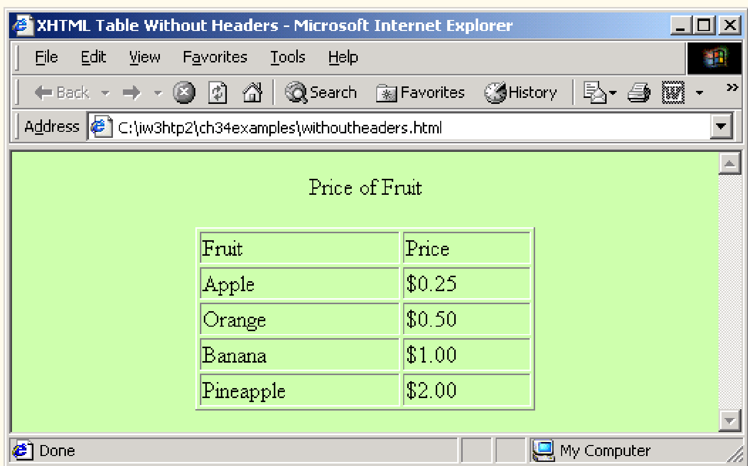


Fig. 25.3 XHTML table without accessibility modifications (part 2 of 2).

This reading does not present the content of the table adequately. WCAG 1.0 recommends using CSS instead of tables, unless the tables' content linearizes in an understandable manner.

If the table in Fig. 25.3 were large, the screen reader's linearized reading would be even more confusing to users. By modifying the **<td>** tag with the **headers** attribute and modifying *header cells* (cells specified by the **<th>** tag) with the **id** attribute, a table will be read as intended. Figure 25.4 demonstrates how these modifications change the way a table is interpreted.

This table does not appear to be different from a standard XHTML table. However, the table is read in a more intelligent manner, when using a screen reader. A screen reader vocalizes the data from the table in Fig. 25.4 as follows:

```
Caption: Price of Fruit
Summary: This table uses th and the id and headers attributes
to make the table readable by screen readers.
Fruit: Apple, Price: $0.25
Fruit: Orange, Price: $0.50
Fruit: Banana, Price: $1.00
Fruit: Pineapple, Price: $2.00
```

Every cell in the table is preceded by its corresponding header when read by the screen reader. This format helps the listener understand the table. The ***headers*** *attribute* is

intended specifically for tables that hold large amounts of data. Most small tables linearize well as long as the **<th>** tag is used properly. The **summary** attribute and the **caption** element are also suggested. For more examples demonstrating how to make tables accessible, visit **www.w3.org/TR/WCAG**.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 25.4: withheaders.html -->
<!-- Table with headers          -->

<html>
   <head>
      <title>XHTML Table With Headers</title>

      <style type = "text/css">
         body { background-color: #ccffaa;
                text-align: center }
      </style>
   </head>

   <body>

   <!-- this table uses the id and headers attributes to   -->
   <!-- ensure readability by text-based browsers. It also -->
   <!-- uses a summary attribute, used screen readers to   -->
   <!-- describe the table                                 -->

      <table width = "50%" border = "1"
         summary = "This table uses th elements and id and
         headers attributes to make the table readable
         by screen readers">

         <caption><strong>Price of Fruit</strong></caption>

         <tr>
            <th id = "fruit">Fruit</th>
            <th id = "price">Price</th>
         </tr>

         <tr>
            <td headers = "fruit">Apple</td>
            <td headers = "price">$0.25</td>
         </tr>

         <tr>
            <td headers = "fruit">Orange</td>
            <td headers = "price">$0.50</td>
         </tr>

```

Fig. 25.4 Table optimized for screen reading using attribute **headers** (part 1 of 2).

```
            <tr>
               <td headers = "fruit">Banana</td>
               <td headers = "price">$1.00</td>
            </tr>

            <tr>
               <td headers = "fruit">Pineapple</td>
               <td headers = "price">$2.00</td>
            </tr>

         </table>

      </body>
   </html>
```

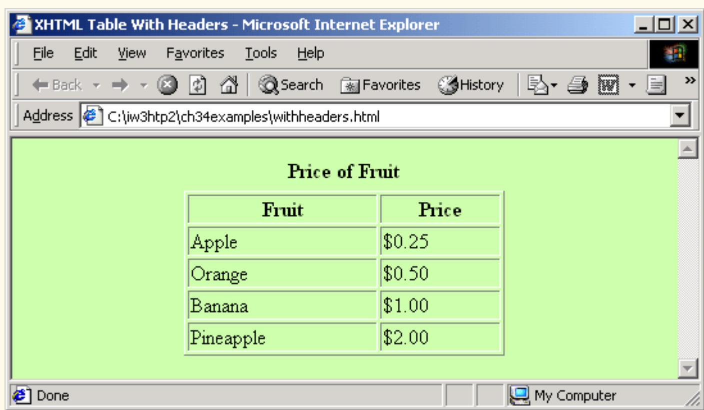


Fig. 25.4 Table optimized for screen reading using attribute **`headers`** (part 2 of 2).

25.7 Accessibility in XHTML Frames

Web designers often use frames to display more than one XHTML file in a single browser window. Frames are a convenient way to ensure that certain content always displays on the screen. Unfortunately, frames often lack proper descriptions, which prevents users with text-based browsers, or users listening with speech synthesizers, from navigating the Web site.

A site with frames must have meaningful descriptions in the **`<title>`** tag for each frame. Examples of good titles include "*Navigation Frame*" and "*Main Content Frame*." Users with text-based browsers, such as Lynx, must choose which frame they want to open; descriptive titles make this choice simpler. However, assigning titles to frames does not solve all the navigation problems associated with frames. The **`<noframes>`** tag allows Web designers to offer alternative content for browsers that do not support frames.

Good Programming Practice 25.1

Always provide titles for frames to ensure that user agents which do not support frames have alternatives.

Good Programming Practice 25.2

Include a title for each frame's contents with the **frame** *element, and, if possible, provide links to the individual pages within the frameset so that users still can navigate through the Web pages. To provide access to browsers that do not support frames, use the* **<noframes>** *tag. It also provides better access to browsers that have limited support.*

WCAG 1.0 suggests using Cascading Style Sheets (CSS) as an alternative to frames, because CSS provides similar functionality and are highly customizible. Unfortunately, the ability to display multiple XHTML documents in a single browser window requires the complete support of HTML 4, which is not widespread. However, the second generation of Cascading Style Sheets (CSS2) displays a single document as if it were several documents. However, CSS2 is not yet fully supported by many user agents.

25.8 Accessibility in XML

XML allows developers to create new markup languages, which may not necessarily incorporate accessibility features. To prevent the proliferation of inaccessible languages, the WAI is developing guidelines—the *XML Guidelines (XML GL)*—for creating accessible XML documents. The XML GL recommends including a text description, similar to XHTML's **<alt>** tag, for each non-text object on a page. To facilitate accessibility further, element types should allow grouping and classification and should identify important content. Without an accessible user interface, other efforts to implement accessibility are less effective, so it is essential to create XSLT or CSS style sheets that can produce multiple outputs, including document outlines.

Many XML languages, including Synchronized Multimedia Integration Language (SMIL) and Scalable Vector Graphics (SVG), implement several of the WAI guidelines. The WAI XML Accessibility Guidelines can be found at **www.w3.org/WAI/PF/xmlgl.htm**.

25.9 Using Voice Synthesis and Recognition with VoiceXML™

A joint effort by AT&T®, IBM®, Lucent™ and Motorola® has created an XML vocabulary that marks up information for *speech synthesizers*, which enable computers to speak to users. This technology, called *VoiceXML*, has tremendous implications for people with visual impairments and for the illiterate. VoiceXML-enabled applications read Web pages to the user and understand words spoken into a microphone through *speech recognition* technology. An example of a speech recognition tool is IBM's *ViaVoice* (**www-4.ibm.com/software/speech**).

A VoiceXML interpreter and VoiceXML browser process VoiceXML, a platform-independent XML-based technology. Web browsers may incorporate these interpreters in the future. When a VoiceXML document is loaded, a *voice server* sends a message to the VoiceXML browser and begins a conversation between the user and the computer.

IBM *WebSphere Voice Server SDK 1.5* is a VoiceXML interpreter that tests VoiceXML documents on a desktop computer. To download the VoiceServer SDK, visit **www.alphaworks.ibm.com/tech/voiceserversdk**. [*Note*: To run the VoiceXML program in Fig. 25.5, download *Java 2 Platform Standard Edition* (Java SDK)

1.3 from **www.java.sun.com/j2se/1.3**. To obtain installation instructions for the VoiceServer SDK and the Java SDK, visit the Deitel & Associates, Inc. Web site at **www.deitel.com**.]

Figure 25.5and Fig. 25.6 show examples of VoiceXML that would be appropriate for a Web site. The document's text is spoken to the user, and the text embedded in the VoiceXML tags allows for interactivity between the user and the browser. The output included in Fig. 25.6 demonstrates a conversation that might take place between a user and a computer after loading this document.

```
<?xml version = "1.0"?>
<vxml version = "1.0">

<!-- Fig. 25.5: main.vxml -->
<!-- Voice page           -->

<link next = "#home">
   <grammar>home</grammar>
</link>

<link next = "#end">
   <grammar>exit</grammar>
</link>

<var name = "currentOption" expr = "'home'"/>

<form>
   <block>
      <emp>Welcome</emp> to the voice page of Deitel and
      Associates. To exit any time say exit.
      To go to the home page any time say home.
   </block>
   <subdialog src = "#home"/>
</form>

<menu id = "home">
   <prompt count = "1" timeout = "10s">
      You have just entered the Deitel home page.
      Please make a selection by speaking one of the
      following options:
      <break msecs = "1000" />
      <enumerate/>
   </prompt>

   <prompt count = "2">
      Please say one of the following.
      <break msecs = "1000" />
      <enumerate/>
   </prompt>

   <choice next = "#about">About us</choice>
   <choice next = "#directions">Driving directions</choice>
```

Fig. 25.5 Home page written in VoiceXML (part 1 of 3).

```
   <choice next = "publications.vxml">Publications</choice>
</menu>

<form id = "about">
   <block>
   About Deitel and Associates, Inc.
   Deitel and Associates, Inc. is an internationally
   recognized corporate training and publishing organization,
   specializing in programming languages, Internet and World
   Wide Web technology and object technology education.
   Deitel and Associates, Inc. is a member of the World Wide
   Web Consortium. The company provides courses on Java, C++,
   Visual Basic, C, Internet and World Wide Web programming
   and Object Technology.
     <assign name = "currentOption" expr = "'about'"/>
     <goto next = "#repeat"/>
   </block>
</form>

<form id = "directions">
  <block>
     Directions to Deitel and Associates, Inc.
     We are located on Route 20 in Sudbury,
     Massachusetts, equidistant from route
   <sayas class = "digits">128</sayas> and route
   <sayas class = "digits">495</sayas>.
   <assign name = "currentOption" expr = "'directions'"/>
   <goto next = "#repeat"/>
  </block>
</form>

<form id = "repeat">
   <field name = "confirm" type = "boolean">
      <prompt>
         To repeat say yes. To go back to home, say no.
      </prompt>

      <filled>
         <if cond = "confirm == true">
            <goto expr = "'#' + currentOption"/>
         <else/>
           <goto next = "#home"/>
         </if>
      </filled>

   </field>
</form>

<form id = "end">
   <block>
      Thank you for visiting Deitel and Associates voice page.
      Have a nice day.
      <exit/>
```

Fig. 25.5 Home page written in VoiceXML (part 2 of 3).

```
96      </block>
97   </form>
98
99   </vxml>
```

Fig. 25.5 Home page written in VoiceXML (part 3 of 3).

A VoiceXML document contains a series of dialogs and subdialogs, which result in spoken interaction between the user and the computer. The **<form>** and **<menu>** tags implement the dialogs. A ***form*** element presents information and gathers data from the user. A ***menu*** element provides users with options and transfers control to other dialogs, based on users' selections.

Lines 7–9 use element ***link*** to create an active link to the home page. Attribute ***next*** specifies the URI navigated to when the link is selected. Element ***grammar*** marks up the text that the user must speak to select the link. In the **link** element, we navigate to the element with **id home** when users speak the word **home**. Lines 11–13 use element **link** to create a link to **id end** when users speak the word **exit**.

Lines 17–24 create a form dialog using element **form**, which collects information from the user. Lines 18–22 present introductory text. Element **block**, which can exist only within a **form** element, groups elements that perform an action or an event. Element **emp** states that a section of text should be spoken with emphasis. If the level of emphasis is not specified, then the default level—*moderate*—is used. Our example uses the default level. [*Note*: To specify an emphasis level, use the **level** attribute. This attribute accepts the following values: *strong*, *moderate*, *none* and *reduced*.]

```
1   <?xml version = "1.0"?>
2   <vxml version = "1.0">
3
4   <!-- Fig. 25.6: publications.vxml        -->
5   <!-- Voice page for various publications -->
6
7   <link next = "main.vxml#home">
8      <grammar>home</grammar>
9   </link>
10  <link next = "main.vxml#end">
11     <grammar>exit</grammar>
12  </link>
13  <link next = "#publication">
14     <grammar>menu</grammar>
15  </link>
16
17  <var name = "currentOption" expr = "'home'"/>
18
19  <menu id = "publication">
20
21     <prompt count = "1" timeout = "12s">
22        Following are some of our publications. For more
23        information visit our web page at www.deitel.com.
24        To repeat the following menu, say menu at any time.
```

Fig. 25.6 Publication page of Deitel's VoiceXML page (part 1 of 4).

```
        Please select by saying one of the following books:

       <break msecs = "1000" />
       <enumerate/>
     </prompt>

     <prompt count = "2">
       Please select from the following books.
       <break msecs = "1000" />
       <enumerate/>
     </prompt>

     <choice next = "#java">Java.</choice>
     <choice next = "#c">C.</choice>
     <choice next = "#cplus">C plus plus.</choice>
  </menu>

  <form id = "java">
     <block>
         Java How to program, third edition.
         The complete, authoritative introduction to Java.
         Java is revolutionizing software development with
         multimedia-intensive, platform-independent,
         object-oriented code for conventional, Internet,
         Intranet and Extranet-based applets and applications.
         This Third Edition of the world's most widely used
         university-level Java textbook carefully explains
         Java's extraordinary capabilities.
      <assign name = "currentOption" expr = "'java'"/>
      <goto next = "#repeat"/>
     </block>
  </form>

  <form id = "c">
     <block>
        C How to Program, third edition.
        This is the long-awaited, thorough revision to the
        world's best-selling introductory C book! The book's
        powerful "teach by example" approach is based on
        more than 10,000 lines of live code, thoroughly
        explained and illustrated with screen captures showing
        detailed output.World-renowned corporate trainers and
        best-selling authors Harvey and Paul Deitel offer the
        most comprehensive, practical introduction to C ever
        published with hundreds of hands-on exercises, more
        than 250 complete programs written and documented for
        easy learning, and exceptional insight into good
        programming practices, maximizing performance, avoiding
        errors, debugging, and testing. New features include
        thorough introductions to C++, Java, and object-oriented
        programming that build directly on the C skills taught
        in this book; coverage of graphical user interface
        development and C library functions; and many new,
        substantial hands-on projects.For anyone who wants to
```

Fig. 25.6 Publication page of Deitel's VoiceXML page (part 2 of 4).

```
        learn C, improve their existing C skills, and understand
        how C serves as the foundation for C++, Java, and
        object-oriented development.
       <assign name = "currentOption" expr = "'c'"/>
       <goto next = "#repeat"/>
    </block>
</form>

<form id = "cplus">
    <block>
        The C++ how to program, second edition.
        With nearly 250,000 sold, Harvey and Paul Deitel's C++
        How to Program is the world's best-selling introduction
        to C++ programming. Now, this classic has been thoroughly
        updated! The new, full-color Third Edition has been
        completely revised to reflect the ANSI C++ standard, add
        powerful new coverage of object analysis and design with
        UML, and give beginning C++ developers even better live
        code examples and real-world projects. The Deitels' C++
        How to Program is the most comprehensive, practical
        introduction to C++ ever published with hundreds of
        hands-on exercises, roughly 250 complete programs written
        and documented for easy learning, and exceptional insight
        into good programming practices, maximizing performance,
        avoiding errors, debugging, and testing. This new Third
        Edition covers every key concept and technique ANSI C++
        developers need to master: control structures, functions,
        arrays, pointers and strings, classes and data
        abstraction, operator overloading, inheritance, virtual
        functions, polymorphism, I/O, templates, exception
        handling, file processing, data structures, and more. It
        also includes a detailed introduction to Standard
        Template Library containers, container adapters,
        algorithms, and iterators.
       <assign name = "currentOption" expr = "'cplus'"/>
       <goto next = "#repeat"/>
    </block>
</form>

<form id = "repeat">
    <field name = "confirm" type = "boolean">

        <prompt>
            To repeat say yes. Say no, to go back to home.
        </prompt>

        <filled>
            <if cond = "confirm == true">
                <goto expr = "'#' + currentOption"/>
            <else/>
                <goto next = "#publication"/>
            </if>
        </filled>
```

Fig. 25.6 Publication page of Deitel's VoiceXML page (part 3 of 4).

```
131       </field>
132    </form>
133 </vxml>
```

```
Computer:
Welcome to the voice page of Deitel and Associates. To exit any time
say exit. To go to the home page any time say home.

User:
Home

Computer:
You have just entered the Deitel home page. Please make a selection by
speaking one of the following options: About us, Driving directions,
Publications.

User:
Driving directions

Computer:
Directions to Deitel and Associates, Inc.
We are located on Route 20 in Sudbury,
Massachusetts, equidistant from route 128
and route 495.
To repeat say yes. To go back to home, say no.
```

Fig. 25.6 Publication page of Deitel's VoiceXML page (part 4 of 4).

The **menu** element on line 26 enables users to select the page to which they would like to link. The **choice** element, which is always part of either a **menu** or a **form**, presents the options. The **next** attribute indicates the page to be loaded when a user makes a selection. The user selects a **choice** element by speaking the text marked up between the tags into a microphone. In this example, the first and second **choice** elements on lines 41–42 transfer control to a *local dialog* (i.e., a location within the same document) when they are selected. The third **choice** element transfers the user to the document **publications.vxml**. Lines 27–33 use element ***prompt*** to instruct the user to make a selection. Attribute ***count*** maintains the number of times a prompt is spoken (i.e., each time a prompt is read, **count** increments by one). The **count** attribute transfers control to another prompt once a certain limit has been reached. Attribute **timeout** specifies how long the program should wait after outputting the prompt for users to respond. In the event that the user does not respond before the timeout period expires, lines 35–39 provide a second, shorter prompt to remind the user to make a selection.

When the user chooses the **publications** option, the **publications.vxml** (Fig. 25.6) loads into the browser. Lines 106–111 define **link** elements that provide links to **main.vxml**. Lines 112–114 provide links to the **menu** element (lines 118–138), which asks users to select one of the publications: Java, C or C++. The **form** elements on lines 140–214 describe each of the books on these topics. Once the browser speaks the description, control transfers to the **form** element with an **id** attribute that has a value equal to **repeat** (lines 216–231).

Figure 25.6 provides a brief description of each VoiceXML tag used in the previous example (Fig. 25.6).

VoiceXML Tag	Description
`<assign>`	Assigns a value to a variable.
`<block>`	Presents information to users without any interaction between the user and the computer (i.e., the computer does not expect any input from the user).
`<break>`	Instructs the computer to pause its speech output for a specified period of time.
`<choice>`	Specifies an option in a `menu` element.
`<enumerate>`	Lists all the available options to the user.
`<exit>`	Exits the program.
`<filled>`	Contains elements to be executed when the computer receives user input for a `form` element.
`<form>`	Gathers information from the user for a set of variables.
`<goto>`	Transfers control from one dialog to another.
`<grammar>`	Specifies grammar for the expected input from the user.
`<if>`, `<else>`, `<elseif>`	Control statements used for making logic decisions.
`<link>`	A transfer of control similar to the `goto` statement, but a `link` can be executed at any time during the program's execution.
`<menu>`	Provides user options and transfers control to other dialogs, based on the selected option.
`<prompt>`	Specifies text to be read to the user when a selection is needed.
`<subdialog>`	Calls another dialog. After executing the subdialog, the calling dialog resumes control.
`<var>`	Declares a variable.
`<vxml>`	The top-level tag specifying that the document should be processed by a VoiceXML interpreter.

Fig. 25.7 Some VoiceXML tags.

25.10 CallXML™

Another advancement in voice technology for people with visual impairments is *CallXML*, a technology created and supported by *Voxeo* (**www.voxeo.com**). CallXML creates phone-to-Web applications that control incoming and outgoing telephone calls. Some examples of CallXML applications include voice mail, interactive voice response systems and Internet call waiting. While VoiceXML assists individuals with visual impairments by reading Web pages, CallXML provides individuals with visual impairments access to Web-based content through telephones.

When users access CallXML applications, a *text-to-speech (TTS)* engine reads information contained within CallXML elements. A TTS engine converts text to an automated voice. Web applications respond to the caller's input. [*Note*: A touch-tone phone is required to access CallXML applications.]

Typically, CallXML applications play pre-recorded audio clips or text as output, requesting a response as input. An audio clip may contain a greeting that introduces callers to the application or to a menu of options that requires callers to make touch-tone entries. Certain applications, such as voice mail, may require verbal and touch-tone input. Once the input is received, the application responds by invoking CallXML elements such as **`text`**, which contains the information a TTS engine reads to users. If the application does not receive input within a designated time frame, it prompts the user to enter valid input.

When a user accesses a CallXML application, the incoming telephone call is referred to as a *session*. A CallXML application can support multiple sessions, enabling the application to receive multiple telephone calls simultaneously. Each session is independent of the others and is assigned a unique *sessionID* for identification. A session terminates either when the user hangs up the telephone or when the CallXML application invokes the **`hangup`** element. Our first CallXML example shows the classic **Hello World** example (Fig. 25.8).

```
<?xml version = "1.0" encoding = "UTF-8"?>

<!-- Fig. 25.8: hello.xml             -->
<!-- The classic Hello World example -->

<callxml>
   <text>Hello World.</text>
</callxml>
```

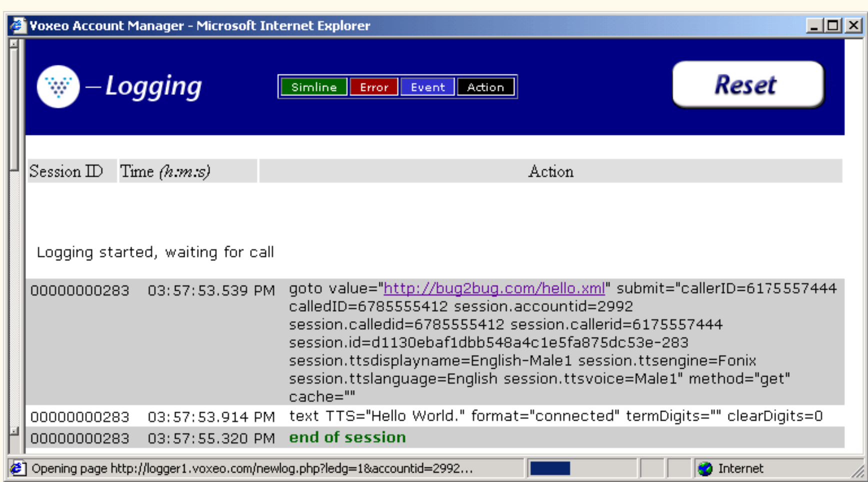


Fig. 25.8 **Hello World** CallXML example.

Line 1 contains the optional *XML declaration*.Value ***version*** indicates the XML version to which the document conforms. Currently, this is **version = 1.0**. Value ***encoding*** indicates the type of *Unicode* encoding to use. This example uses UTF-8, which requires eight bits to transfer and receive data. More information on Unicode may be found in Appendix G, Unicode®.

The **<callxml>** tag on line 6 declares the contents of a CallXML document. Line 7 contains the **Hello World text**. All text spoken by a text-to-speech (TTS) engine needs to reside within **<text>** tags.

To deploy a CallXML application, register with the *Voxeo* Community (**community.voxeo.com**), a Web resource for creating, debugging and deploying phone applications. For the most part, Voxeo is a free Web resource. However, the company charges fees when CallXML applications are deployed commercially. The Voxeo Community assigns a unique telephone number to each CallXML application so that external users may access and interact with the application. [*Note*: Voxeo assigns telephone numbers to applications that reside on the Internet. If you have access to a Web server (IIS, PWS, Apache, etc.), use it to post your CallXML application. Otherwise, open an Internet account using one of the many Internet-service companies (e.g., **www.geocities.com**, **www.angelfire.com**). These companies allow you to post documents on the Internet by using their Web servers.]

Figure 25.8 demonstrates the *logging* feature of the **Voxeo Account Manager**, which is accessible to registered members. The logging feature records and displays the "conversation" between the user and the application. The first row of the logging feature displays the URL of the CallXML application and the *global variables* associated with each session. The application (program) creates and assigns values to global variables, which the entire application can access and modify, at the start of each session. The subsequent row(s) display(s) the "conversation." This example shows a one-way conversation (because the application does not accept any input from the user) in which the TTS says **Hello World**. The last row shows the **end of session** message, which states that the phone call has terminated. The logging feature assists developers in debugging their applications. By observing the "conversation," a developer can determine at which point the application terminates. If the application terminates abruptly ("crashes"), the logging feature states the type and location of the error, so that a developer knows the particular section of the application on which to focus.

The next example (Fig. 25.9) shows a CallXML application that reads the ISBN values of three Deitel textbooks—*Internet and World Wide Web How to Program: Second Edition*, *XML How to Program* and *Java How to Program: Fourth Edition*—based on the user's touch-tone input. [*Note*: The following code has been formatted for presentation purposes.]

```
<?xml version = "1.0" encoding = "UTF-8"?>

<!-- Fig. 25.9: isbn.xml                        -->
<!-- Reads the ISBN value of three Deitel books -->

<callxml>
```

Fig. 25.9 CallXML example that reads three ISBN values (part 1 of 3).

```
      <block>
         <text>
            Welcome. To obtain the ISBN of the Internet and World
            Wide Web How to Program: Second Edition, please enter 1.
            To obtain the ISBN of the XML How to Program,

            please enter 2. To obtain the ISBN of the Java How
            to Program: Fourth Edition, please enter 3. To exit the
            application, please enter 4.
         </text>

         <!-- obtains the numeric value entered by the user and -->
         <!-- stores it in the variable ISBN. The user has 60   -->
         <!-- seconds to enter one numeric value                -->
         <getDigits var = "ISBN"
            maxDigits = "1"
            termDigits = "1234"
            maxTime = "60s" />

         <!-- requests that the user enter a valid numeric -->
         <!-- value after the elapsed time of 60 seconds   -->
         <onMaxSilence>
            <text>
               Please enter either 1, 2, 3 or 4.
            </text>

            <getDigits var = "ISBN"
               termDigits = "1234"
               maxDigits = "1"
               maxTime = "60s" />

         </onMaxSilence>

         <onTermDigit value = "1">
            <text>
               The ISBN for the Internet book is 0130308978.
               Thank you for calling our CallXML application.
               Good-bye.
            </text>
         </onTermDigit>

         <onTermDigit value = "2">
            <text>
               The ISBN for the XML book is 0130284173.
               Thank you for calling our CallXML application.
               Good-bye.
            </text>
         </onTermDigit>

         <onTermDigit value = "3">
            <text>
               The ISBN for the Java book is 0130341517.
               Thank you for calling our CallXML application.
               Good-bye.
```

Fig. 25.9 CallXML example that reads three ISBN values (part 2 of 3).

```
60             </text>
61          </onTermDigit>
62
63          <onTermDigit value = "4">

64             <text>
65                Thank you for calling our CallXML application.
66                Good-bye.
67             </text>
68          </onTermDigit>
69       </block>
70
71       <!-- event handler that terminates the call -->
72       <onHangup />
73    </callxml>
```

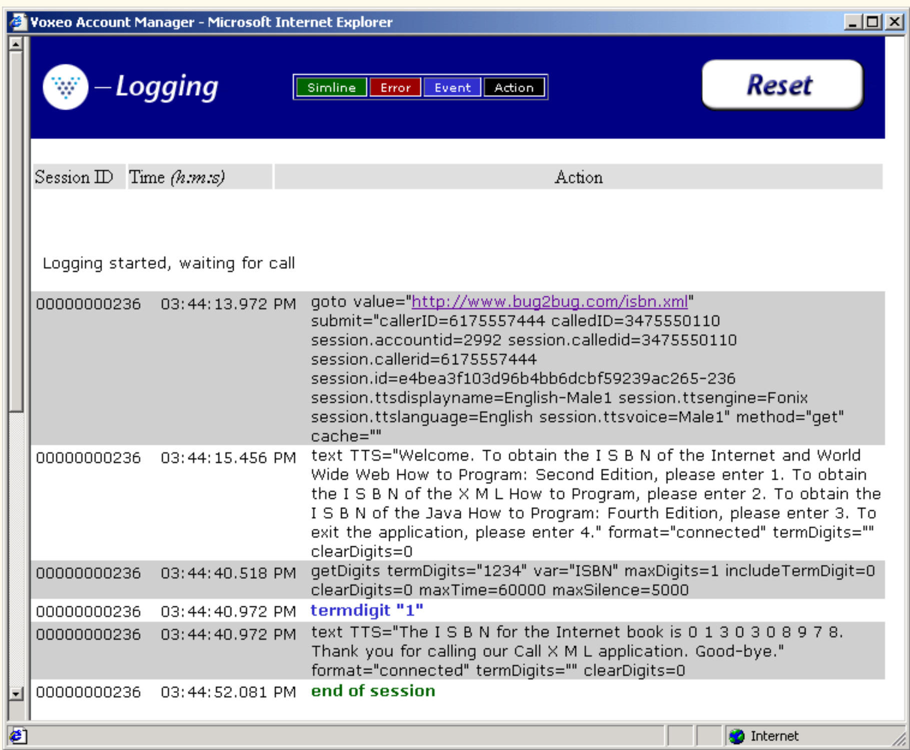


Fig. 25.9 CallXML example that reads three ISBN values (part 3 of 3).

The ***<block>*** tag (line 7) encapsulates other CallXML tags. Usually, CallXML tags that perform a similar task should be enclosed within **<block>**...**</block>**. The **block** element in this example encapsulates the ***<text>***, ***<getDigits>***, ***<onMaxSilence>*** and ***<onTermDigit>*** tags. A **block** element can contain nested **block** elements.

Lines 20–23 show some attributes of the **<getDigits>** tag. The **getDigits** element obtains the user's touch-tone response and stores it in the variable declared by the

var attribute (i.e., **ISBN**). The ***maxDigits*** attribute (line 21) indicates the maximum number of digits that the application can accept. This application accepts only one character. If no number is stated, then the application uses the default value—*nolimit*.

The ***termDigits*** attribute (line 22) contains the list of characters that terminate user input. When a character from this list is received as input, the CallXML application is notified that the last acceptable input has been received and that any character entered after this point is invalid. These characters do not terminate the call; they simply notify the application to proceed to the next step because the necessary input has been received. In our example, the values for **termDigits** are one, two, three or four. The default value for **termDigits** is the null value (**""**).

The ***maxTime*** attribute (line 23) indicates the maximum amount of time to wait for a user response (i.e., 60 seconds). If no input is received within the given time frame, then the CallXML application may terminate—a drastic measure. The default value for this attribute is 30 seconds.

The ***onMaxSilence*** element (lines 27–37) is an *event handler* that is invoked when the **maxTime** (or **maxSilence**) expires. An event handler notifies the application of the appropriate action to perform. In this case, the application asks the user to enter a value because the **maxTime** has expired. After receiving input, **getDigits** (line 32) stores the value in the **ISBN** variable.

The ***onTermDigit*** element (lines 39–68) is an event handler that notifies the application of the appropriate action to perform when users select one of the **termDigits** characters. At least one **<onTermDigit>** tag must be associated with the **getDigits** element, even if the default value (**""**) is used. We provide four actions that the application can perform depending on the user-entered value. For example, if the user enters **1**, the application reads the ISBN value of the *Internet and World Wide Web How to Program: Second Edition* textbook.

Line 72 contains the ***<onHangup/>*** event handler, which terminates the telephone call when the user hangs up the telephone. Our **<onHangup>** event handler is an empty tag (i.e., there is no action to perform when this tag is invoked).

The logging feature in Fig. 25.9 displays the "conversation" between the application and the user. The first row displays the URL of the application and the global variables of the session. The subsequent rows display the "conversation"—the application asks the caller which ISBN value to read, the caller enters **1** (*Internet and World Wide Web How to Program: Second Edition*) and the application reads the corresponding ISBN. The **end of session** message states that the application has terminated.

Brief descriptions of several logic and action CallXML elements are provided in Fig. 25.10. *Logic elements* assign values to, and clear values from, the session variables, and *action elements* perform specified tasks, such as answering and terminating a telephone call during the current session. A complete list of CallXML elements is available at:

www.oasis-open.org/cover/callxmlv2.html

25.11 JAWS® for Windows

JAWS (Job Access with Sound) is one of the leading screen readers on the market today. Henter-Joyce, a division of Freedom Scientific™, created this application to help people with visual impairments use technology.

To download a demonstration version of JAWS, visit **`www.hj.com/JAWS/JAWS37DemoOp.htm`** and select the **JAWS 3.7 FREE Demo** link. The demo expires after 40 minutes. The computer must be rebooted before another 40-minute session can be started.

Elements	Description
`assign`	Assigns a **`value`** to a variable, **`var`**.
`clear`	Clears the contents of the **`var`** attribute.
`clearDigits`	Clears all digits that the user has entered.
`goto`	Navigates to another section of the current CallXML application or to a different CallXML application. The **`value`** attribute specifies the application URL. The **`submit`** attribute lists the variables that are passed to the invoked application. The **`method`** attribute states whether to use the HTTP *get* or *post* request types when sending and retrieving information. A *get* request retrieves data from a Web server without modifying the contents, while the *post* request sends modified data.
`run`	Starts a new CallXML session for each call. The **`value`** attribute specifies which CallXML application to retrieve. The **`submit`** attribute lists the variables that are passed to the invoked application. The **`method`** attribute states whether to use the HTTP *get* or *post* request type. The **`var`** attribute stores the identification number of the session.
`sendEvent`	Allows multiple sessions to exchange messages. The **`value`** attribute stores the message, and the **`session`** attribute specifies the identification number of the session that receives the message.
`answer`	Answers an incoming telephone call.
`call`	Calls the URL specified by the **`value`** attribute. The **`callerID`** attribute contains the phone number that is displayed on a CallerID device. The **`maxTime`** attribute specifies the length of time to wait for the call to be answered before disconnecting.
`conference`	Connects multiple sessions so that people can participate in a conference call. The **`targetSessions`** attribute specifies the identification numbers of the sessions, and the **`termDigits`** attribute indicates the touch-tone keys that terminate the call.
`wait`	Waits for user input. The **`value`** attribute specifies how long to wait. The **`termDigits`** attribute indicates the touch-tone keys that terminate the **`wait`** element.

Fig. 25.10 List of some CallXML elements (part 1 of 2).

Elements	Description
`play`	Plays an audio file or a value that is stored as a number, date or amount of money and is indicated by the `format` attribute. The `value` attribute contains the information (location of the audio file, number, date or amount of money) that corresponds to the `format` attribute. The `clearDigits` attribute specifies whether or not to delete the previously entered input. The `termDigits` attribute indicates the touch-tone keys that terminate the audio file, etc.
`recordAudio`	Records an audio file and stores it at the URL specified by `value`. The `format` attribute indicates the file extension of the audio clip. Other attributes include `termDigits`, `clearDigits`, `max-Time` and `maxSilence`.

Fig. 25.10 List of some CallXML elements (part 2 of 2).

The JAWS demo is fully functional and includes an extensive, highly customized help system. Users can select which voice to use and the rate at which text is spoken. Users also can create keyboard shortcuts. Although the demo is in English, the full version of JAWS 3.7 allows the user to choose one of several supported languages.

JAWS also includes special key commands for popular programs such as Microsoft Internet Explorer and Microsoft Word. For example, when browsing in Internet Explorer, JAWS' capabilities extend beyond reading the content on the screen. If JAWS is enabled, pressing *Insert* + *F7* in Internet Explorer opens a **Links List** dialog, which displays all the links available on a Web page. For more information about JAWS and the other products offered by Henter-Joyce, visit **`www.hj.com`**.

25.12 Other Accessibility Tools

Many additional accessibility products are available to assist people with disabilities. This section describes a variety of accessibility products, including hardware items and advanced technologies.

A *braille keyboard*, in addition to having each key labeled with the letter it represents, has the equivalent braille symbol printed on the key. Braille keyboards are combined most often with a speech synthesizer or a braille display, so users can interact with the computer to verify that their typing is correct.

Speech synthesis is another research-intensive area that benefits people with disabilities. Speech synthesizers aid those who are unable to communicate verbally. However, the growing popularity of the Web has prompted a great deal of work in the field of speech synthesis and speech recognition. These technologies are allowing individuals with disabilities to use computers more than ever before. The development of speech synthesizers is also enabling the improvement of other technologies, such as VoiceXML and *AuralCSS* (**`www.w3.org/TR/REC-CSS2/aural.html`**). These tools allow people with visual impairment and the illiterate to access Web sites.

Despite the existence of adaptive software and hardware for people with visual impairments, the accessibility of computers and the Internet is still hampered by the high costs, rapid obsolescence and unnecessary complexity of current technology. Moreover, almost all software currently available requires installation by a person who can see. *Ocularis* is a project launched in the open-source community to help address these problems. Open source software for people with visual impairments already exists, and although it is often superior to its proprietary, closed-source counterparts, it has not yet reached its full potential. Ocularis ensures that the blind can use the Linux operating system fully, by providing an *Audio User Interface (AUI)*. Products that integrate with Ocularis include a word processor, calculator, basic finance application, Internet browser and e-mail client. A screen reader will also be included with programs that have a command-line interface. The official Ocularis Web site is located at **`ocularis.sourceforge.net`**.

People with visual impairments are not the only beneficiaries of the effort being made to improve markup languages. People with hearing impairments also have a number of tools to help them interpret auditory information delivered over the Web, such as *Synchronized Multimedia Integration Language* (SMIL™), discussed in Chapter 33, Multimedia. This markup language adds extra *tracks*—layers of content found within a single audio or video file—to multimedia content. The additional tracks can contain closed captioning.

Technologies also are being designed to help people with severe disabilities, such as *quadriplegia*, a form of paralysis that affects the body from the neck down. One such technology, *EagleEyes*, developed by researchers at Boston College (**`www.bc.edu/eagleeyes`**), is a system that translates eye movements into mouse movements. Users move the mouse cursors by moving their eyes or heads and thereby can control computers.

The company CitXCorp is developing new technology that translates Web information through the telephone. Information on a specific topic can be accessed by dialing the designated number. The new software is expected to be made available to users for $10 per month. For more information on regulations governing the design of Web sites to accommodate people with disabilities, visit **`www.access-board.gov`**.

In alliance with Microsoft, GW Micro, Henter-Joyce and Adobe Systems, Inc. are also working on software to aid people with disabilities. JetForm Corp also is accommodating the needs of people with disabilities by developing server-based XML software. The new software allows users to download a format that best meets their needs.

There are many services on the Web that assist e-business owners in designing their Web sites to be accessible to individuals with disabilities. For additional information, the U.S. Department of Justice (**`www.usdoj.gov`**) provides extensive resources detailing legal issues and current technologies related to people with disabilities.

These examples are just a few of the accessibility projects and technologies that currently exist. For more information on Web and general computer accessibility, see the resources provided in Section 25.14, Internet and World Wide Web Resources.

25.13 Accessibility in Microsoft® Windows® 2000

Beginning with Microsoft *Windows 95*, Microsoft has included accessibility features in its operating systems and many of its applications, including *Office 97*, *Office 2000* and *Netmeeting*. In Microsoft *Windows 2000*, the accessibility features have been significantly enhanced. All the accessibility options provided by Windows 2000 are available through the

Accessibility Wizard, which guides users through all the Windows 2000 accessibility features and configures their computers according to the chosen specifications. This section guides users through the configuration of their Windows 2000 accessibility options using the **Accessibility Wizard**.

To access the **Accessibility Wizard**, users must have Microsoft Windows 2000. Select the **Start** button and select **Programs** followed by **Accessories**, **Accessibility** and **Accessibility Wizard**. When the wizard starts, the **Welcome** screen is displayed. Select **Next** to display a dialog (Fig. 25.11) that asks the user to select a font size. Click **Next**.

Figure 25.12 shows the next dialog displayed. This dialog allows the user to activate the font size settings chosen in the previous window, change the screen resolution, enable the *Microsoft Magnifier* (a program that displays an enlarged section of the screen in a separate window) and disable personalized menus (a feature which hides rarely used programs from the start menu, which can be a hindrance to users with disabilities). Make selections and select **Next**.

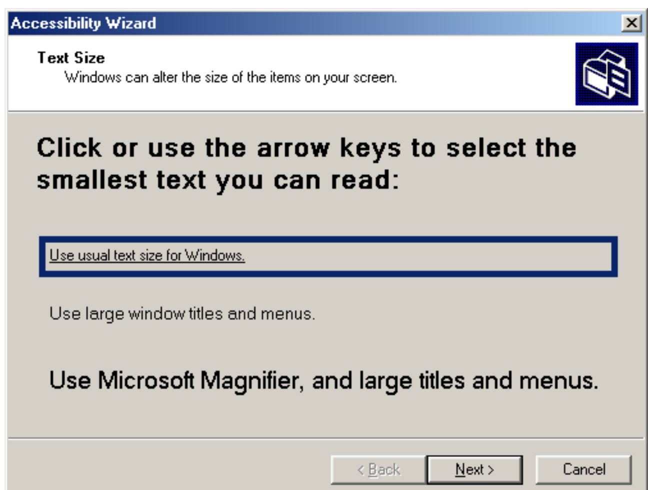


Fig. 25.11 **Text Size** dialog.

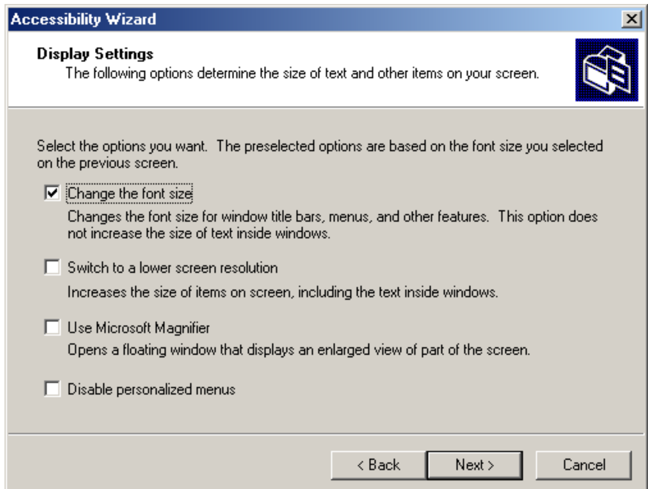


Fig. 25.12 **Display Settings** dialog.

The next dialog (Fig. 25.13) inquires about the user's disabilities, which allows the **Accessibility Wizard** to customize Windows to better suit their needs. We selected everything for demonstration purposes. Select **Next** to continue.

25.13.1 Tools for People with Visual Impairments

When we checked all the options in Fig. 25.13, the wizard configured Windows for people with visual impairments. As shown in Fig. 25.14, the dialog box allows users to resize the scroll bars and window borders to increase their visibility. Select **Next** to proceed to the next dialog.

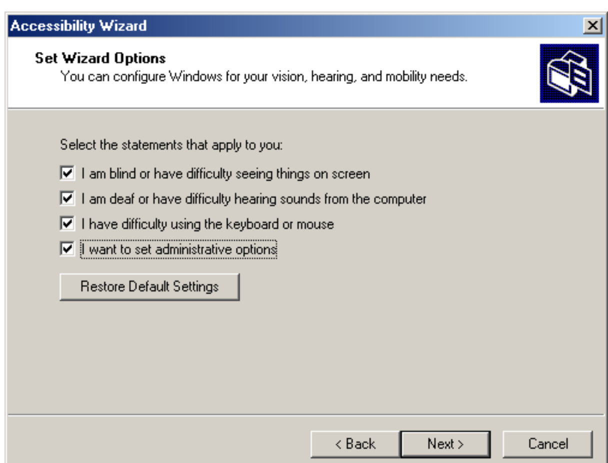


Fig. 25.13 **Accessibility Wizard** initialization options.

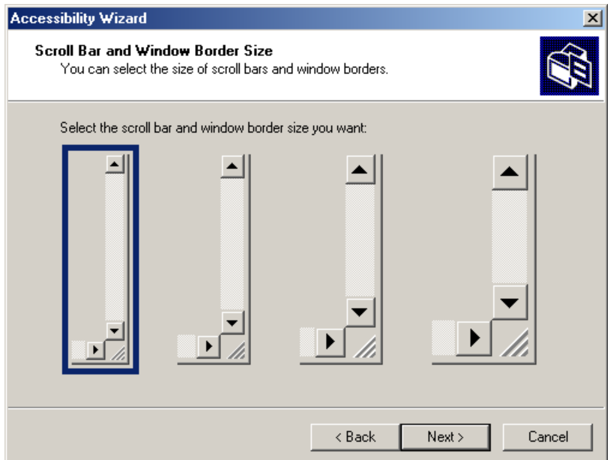


Fig. 25.14 **Scroll Bar and Window Border Size** dialog.

The dialog in Fig. 25.15 allows users to resize icons. Users with poor vision, as well as users who have trouble reading, benefit from large icons.

Selecting **Next** displays the **Display Color Settings** dialog (Fig. 25.16). These settings allow users to change Windows' color schemes and resize various screen elements. Select **Next** to view the dialog (Fig. 25.17) for customizing the mouse cursor.

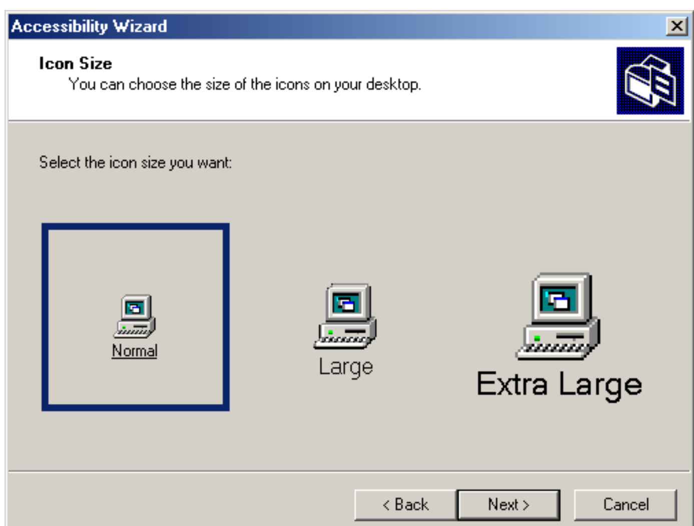


Fig. 25.15 Setting up window element sizes.

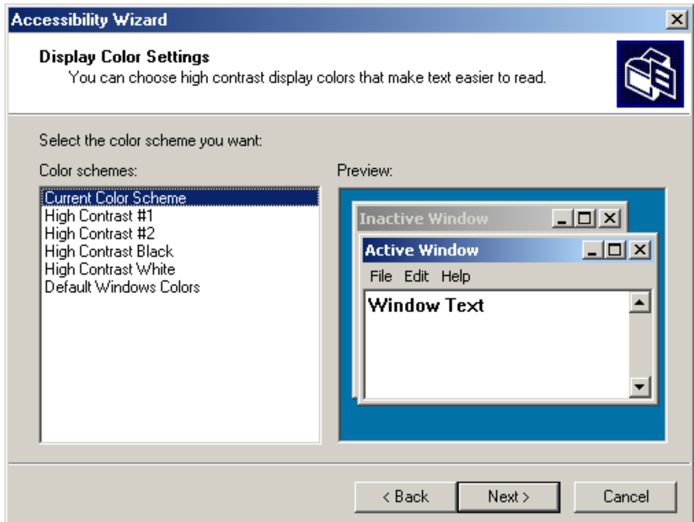


Fig. 25.16 **Display Color Settings** options.

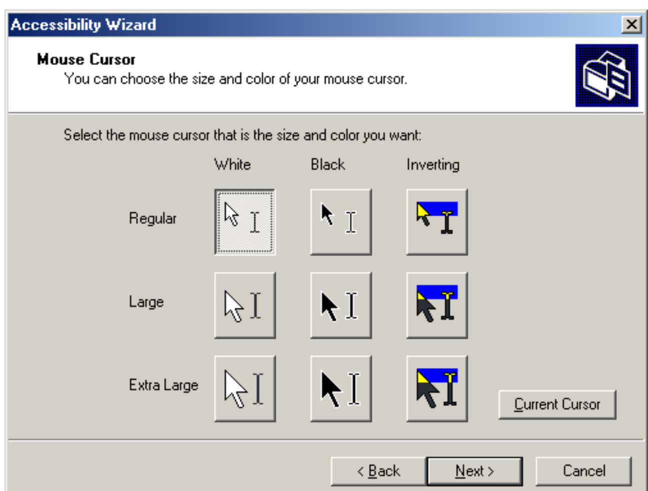


Fig. 25.17 **Accessibility Wizard** mouse cursor adjustment tool.

Anyone who has ever used a laptop computer knows how difficult it is to see the mouse cursor. This is also a problem for people with visual impairments. To help solve this problem, the wizard offers larger cursors, black cursors and cursors that invert the colors of objects underneath them. Select **Next**.

25.13.2 Tools for People with Hearing Impairments

This section, which focuses on accessibility for people with hearing impairments, begins with the ***SoundSentry*** window (Fig. 25.18). **SoundSentry** creates visual signals when system events occur. For example, people with hearing impairments are unable to hear the beeps that normally warn users, so **SoundSentry** flashes the screen when a beep occurs. To continue to the next dialog, select **Next**.

The next window is the ***ShowSounds*** window (Fig. 25.19). **ShowSounds** adds captions to spoken text and other sounds produced by today's multimedia-rich software. For **ShowSounds** to work, software developers must provide the captions and spoken text specifically within their software. Make selections and select **Next**.

25.13.3 Tools for Users Who Have Difficulty Using the Keyboard

The next dialog is **StickyKeys** (Fig. 25.20). ***StickyKeys*** helps users who have difficulty pressing multiple keys at the same time. Many important computer commands are invoked by pressing specific key combinations. For example, the reboot command requires pressing *Ctrl+Alt+Delete* simultaneously. **StickyKeys** allows users to press key combinations in sequence rather than simultaneously. Select **Next** to continue to the ***BounceKeys*** dialog (Fig. 25.21).

Another common problem for certain users with disabilities is accidently pressing the same key more than once. This problem typically results from pressing a key for a long period of time. **BounceKeys** force the computer to ignore repeated keystrokes. Select **Next**.

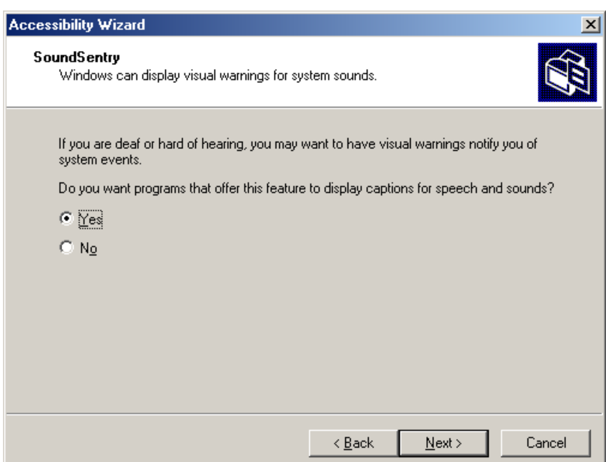


Fig. 25.18 **SoundSentry** dialog.

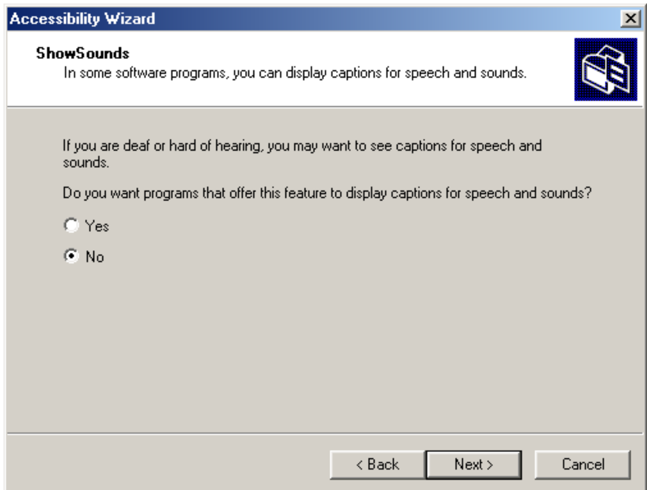


Fig. 25.19 **ShowSounds** dialog.

ToggleKeys (Fig. 25.22) alerts users that they have pressed one of the lock keys (i.e., *Caps Lock*, *Num Lock* and *Scroll Lock*) by sounding an audible beep. Make selections and select **Next**.

The **Extra Keyboard Help** dialog (Fig. 25.23) activates a tool that displays information such as keyboard shortcuts and tool tips when they are available. Like **ShowSounds**, this tool requires that software developers provide the content to be displayed. Selecting **Next** loads the **MouseKeys** (Fig. 25.24) customization window.

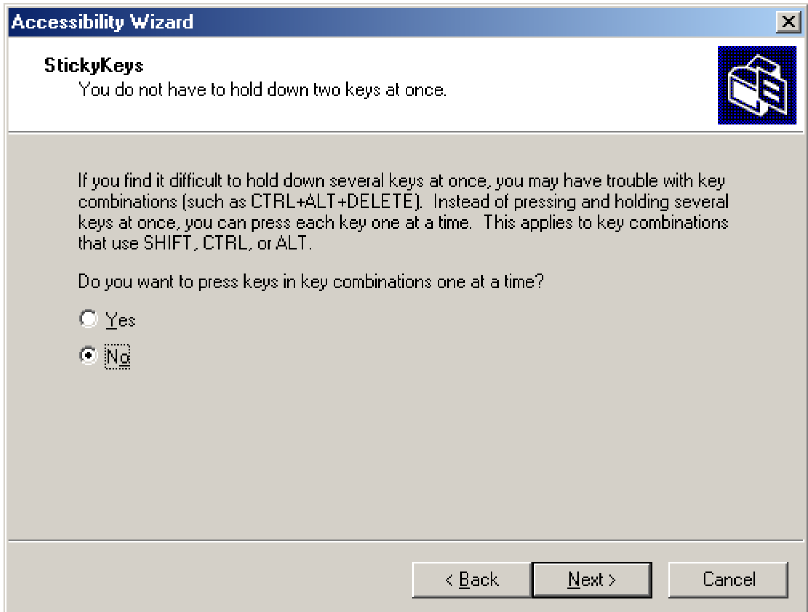


Fig. 25.20 **StickyKeys** window.

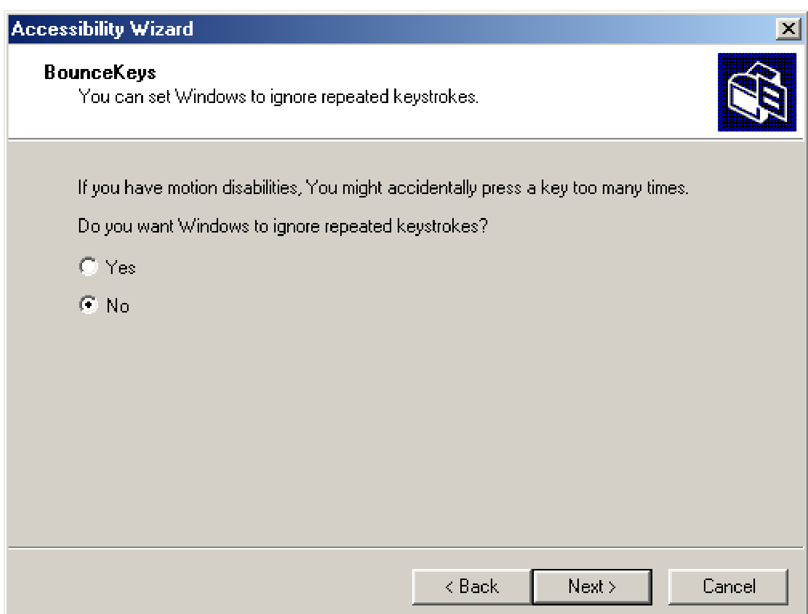


Fig. 25.21 **BounceKeys** dialog.

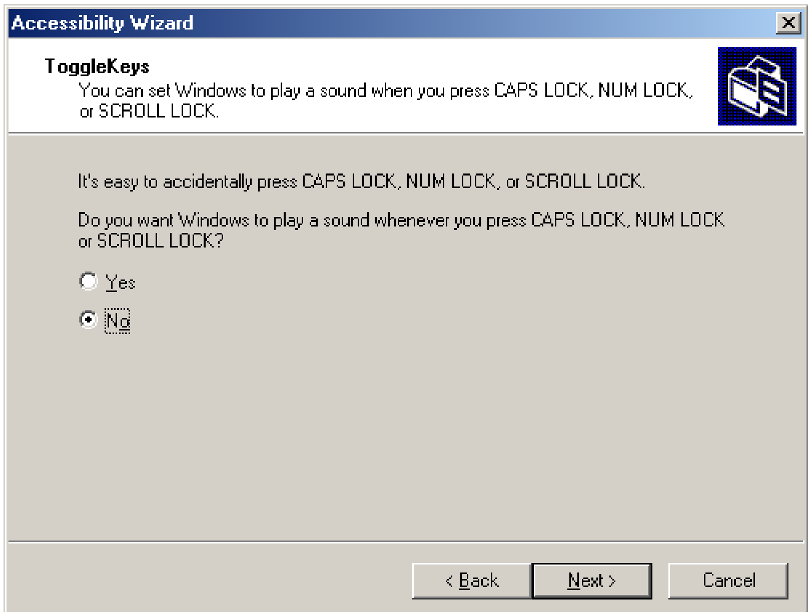


Fig. 25.22 **ToggleKeys** window.

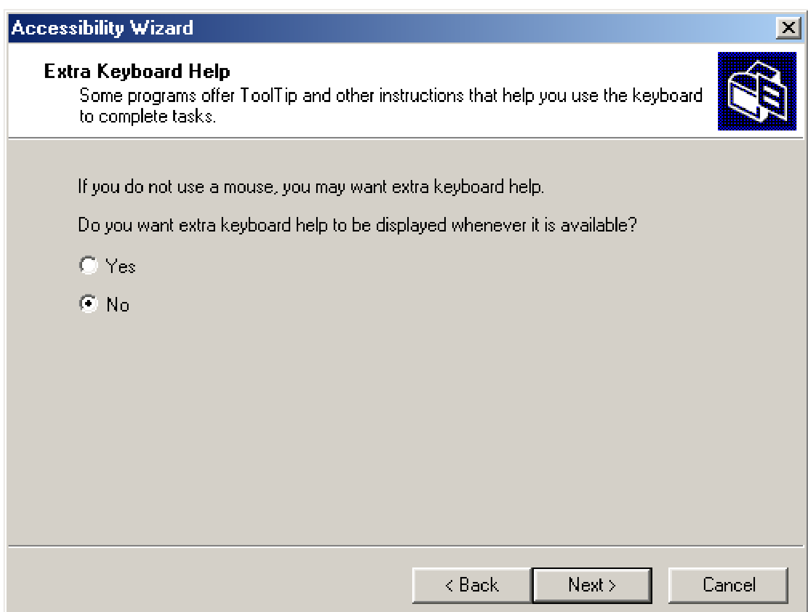


Fig. 25.23 **Extra Keyboard Help** dialog.

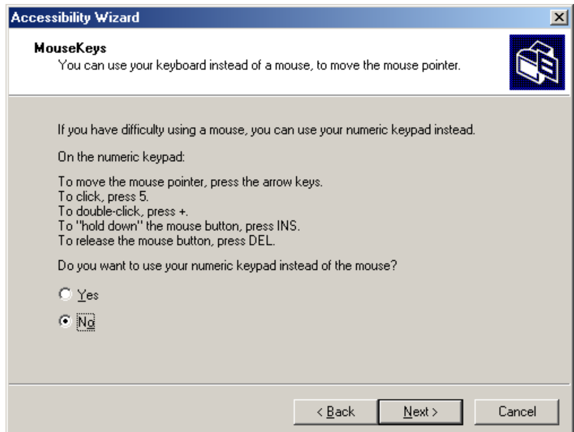


Fig. 25.24 **MouseKeys** window.

MouseKeys uses the keyboard to emulate mouse movements. The arrow keys direct the mouse, while the ***5*** key sends a single click. To double click, the user must press the + key; to simulate holding down the mouse button, the user must press the *Ins* (*Insert*) key and to release the mouse button, the user must press the *Del* (*Delete*) key. To continue to the next screen in the **Accessibility Wizard**, select **Next**.

Today's computer tools are made almost exclusively for right-handed users, including most computer mice. Microsoft recognized this problem and added the ***Mouse Button Settings*** window (Fig. 25.25) to the **Accessibility Wizard**. This tool allows users to create virtual left-handed mice by swapping the button functions. Select **Next**.

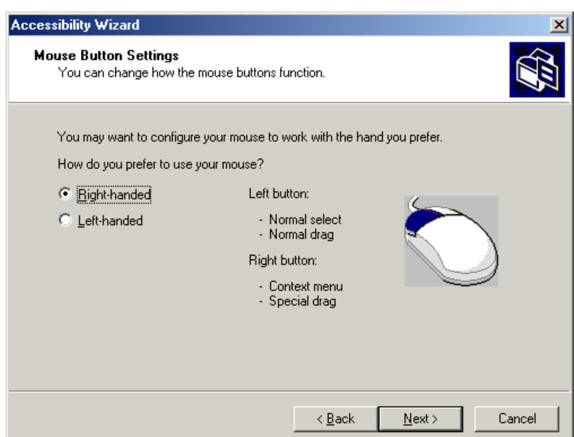


Fig. 25.25 **Mouse Button Settings** window.

Mouse speed is adjusted by using the **MouseSpeed** (Fig. 25.26) dialog of the **Accessibility Wizard**. Dragging the scroll bar changes the speed. Selecting the **Next** button sets the speed and displays the wizard's **Set Automatic Timeouts** window (Fig. 25.27).

Although accessibility tools are important to users with disabilities, they can be a hindrance to users who do not need them. In situations where varying accessibility needs exist, it is important that users be able to turn the accessibility tools on and off as necessary. The ***Set Automatic Timeouts*** window specifies a *timeout* period for the tools. A timeout either enables or disables a certain action after the computer has idled for a specified amount of time. A screen saver is a common example of a program with a timeout period. Here, a timeout is set to toggle the accessibility tools.

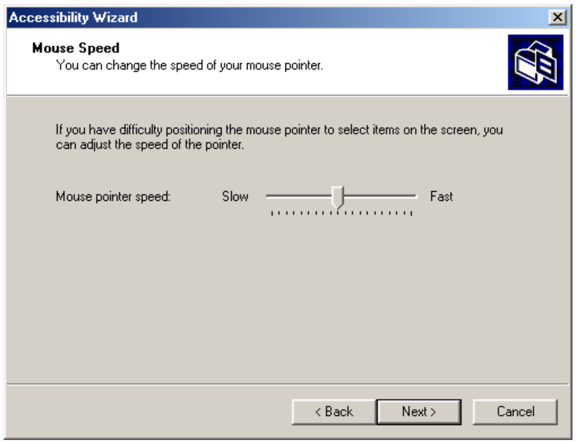


Fig. 25.26 **Mouse Speed** dialog.

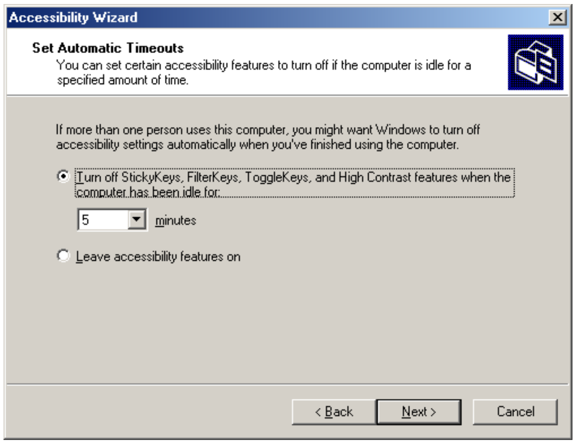


Fig. 25.27 **Set Automatic Timeouts** dialog.

After selecting **Next**, the **Save Settings to File** dialog appears (Fig. 25.28). This dialog determines whether the accessibility settings should be used as the *default settings*, which are loaded when the computer is rebooted, or after a timeout. Set the accessibility settings as the default if the majority of users need them. Users can save the accessibility settings as well, by creating an **.acw** file, which, when clicked, activates the saved accessibility settings on any Windows 2000 computer.

25.13.4 Microsoft Narrator

Microsoft ***Narrator*** is a text-to-speech program for people with visual impairments. It reads text, describes the current desktop environment and alerts users when certain Windows events occur. **Narrator** aids in configuring Microsoft Windows. It is a screen reader that works with Internet Explorer, *Wordpad*, *Notepad* and most programs in the **Control Panel**. Although it is limited outside these applications, **Narrator** is excellent at navigating the Windows environment.

To get an idea of what **Narrator** does, we explain how to use it with various Windows applications. Select the **Start** button and select **Programs**, followed by **Accessories**, **Accessibility** and **Narrator**. Once **Narrator** is open, it describes the current foreground window. It then reads the text inside the window aloud to the user. Selecting **OK** displays the dialog in Fig. 25.29.

Checking the first option instructs **Narrator** to describe menus and new windows when they are opened. The second option instructs **Narrator** to speak the characters entered by the user. The third option moves the mouse cursor to the region being read by **Narrator**. Clicking the **Voice** button enables the user to change the pitch, volume and speed of the narrator voice.

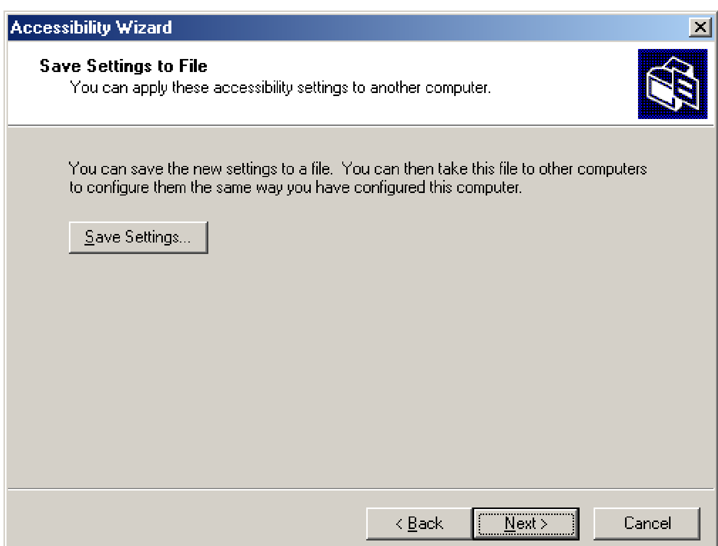


Fig. 25.28 Saving new accessibility settings.

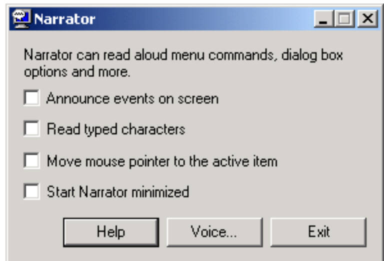


Fig. 25.29 **Narrator** window.

With **Narrator** running, open **Notepad** and select the **File** menu. **Narrator** announces the opening of the program and begins to describe the items in the **File** menu. When scrolling down the list, **Narrator** reads the current item to which the mouse is pointing. Type some text and press *Ctrl-Shift-Enter* to hear **Narrator** read it (Fig. 25.30). If the **Read typed characters** option is checked, **Narrator** reads each character as it is typed. The direction arrows on the keyboard can be used to make **Narrator** read. The up and down arrows cause **Narrator** to speak the lines adjacent to the current mouse position, and the left and right arrows cause **Narrator** to speak the characters adjacent to the current mouse position.

25.13.5 Microsoft On-Screen Keyboard

Some computer users lack the ability to use a keyboard but can use a pointing device such as a mouse. For these users, the *On-Screen Keyboard* is helpful. To access the On-Screen Keyboard, select the **Start** button and select **Programs** followed by **Accessories**, **Accessibility** and **On-Screen Keyboard**. Figure 25.31 shows the layout of the Microsoft On-Screen Keyboard.

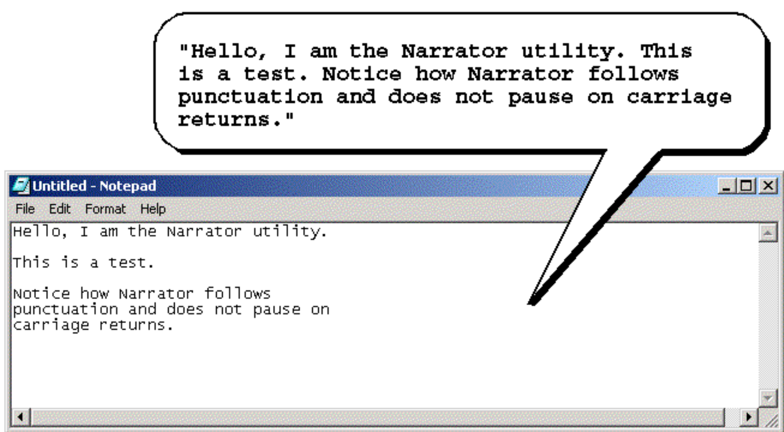


Fig. 25.30 **Narrator** reading **Notepad** text.



Fig. 25.31 Microsoft **On-Screen Keyboard**.

Users who have difficulty using the On-Screen Keyboard should purchase more sophisticated products, such as *Clicker 4*™ by *Inclusive Technology*. Clicker 4 aids people who cannot use a keyboard effectively. Its main feature is its ability to be customized. Keys can have letters, numbers, entire words or even pictures on them. For more information regarding Clicker 4, visit **`www.inclusive.co.uk/catalog/clicker.htm`**.

25.13.6 Accessibility Features in Microsoft Internet Explorer 5.5

Internet Explorer 5.5 offers a variety of options to improve usability. To access IE5.5's accessibility features, launch the program, select the **Tools** menu and select **Internet Options...**. From the **Internet Options** menu, press the button labeled **Accessibility...** to open the accessibility options (Fig. 25.32).

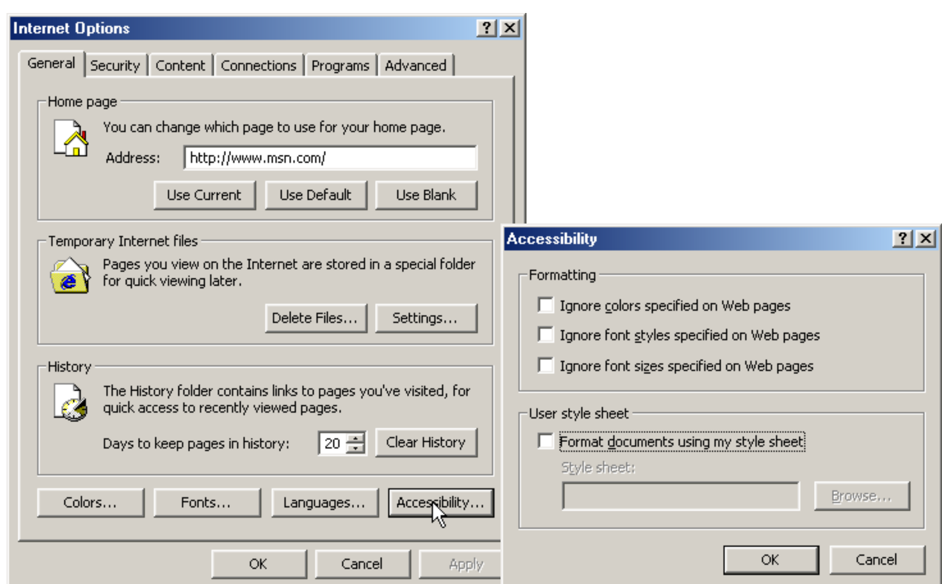


Fig. 25.32 Microsoft Internet Explorer 5.5's accessibility options.

The accessibility options in IE5.5 augment users' Web browsing. Users can ignore Web colors, Web fonts and font size tags. This eliminates problems that arise from poor Web page design and allows users to customize their Web browsers. Users can even specify a *style sheet,* which formats every Web site visited according to users' personal preferences.

These are not the only accessibility options offered in IE5.5. In the **Internet Options** dialog click the **Advanced** tab. This opens the dialog shown in Fig. 25.33. The first option that can be set is labeled **Always expand ALT text for images**. By default, IE5.5 hides some of the **`<alt>`** text if it exceeds the size of the image it describes. This option forces all the text to be shown. The second option reads: **Move system caret with focus/selection changes**. This option is intended to make screen reading more effective. Some screen readers use the *system caret* (the blinking vertical bar associated with editing text) to decide what is read. If this option is not activated, screen readers may not read Web pages correctly.

Web designers often forget to take accessibility into account when creating Web sites and they use fonts that are too small. Many user agents have addressed this problem by allowing the user to adjust the text size. Select the **View** menu and then **Text Size** to change the font size using IE5.5. By default, the text size is set to **Medium**.

25.14 Internet and World Wide Web Resources

There are many accessibility resources on the Internet and World Wide Web, and this section lists a variety of these resources.

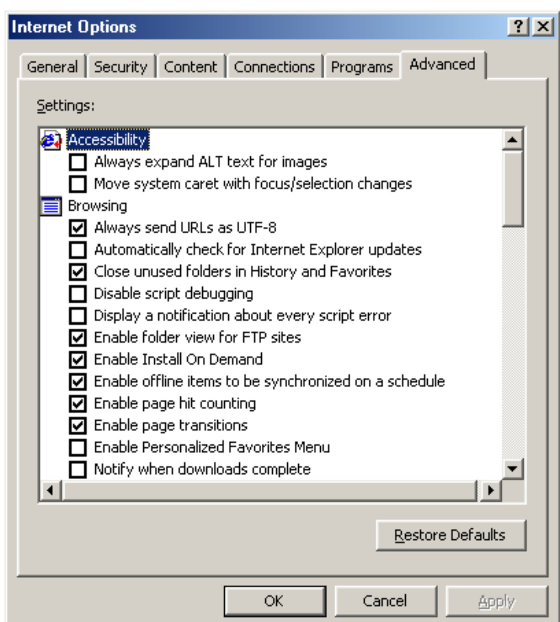


Fig. 25.33 Advanced accessibility settings in Microsoft Internet Explorer 5.5.

www.synapseadaptive.com/joel/natlink.htm
Python module **natlink** allows the user to access and control Dragon NaturallySpeaking, software that provides a speech recognition system for Windows 95/98/NT.

www.w3.org/WAI
The World Wide Web Consortium's *Web Accessibility Initiative (WAI)* site promotes the design of universally accessible Web sites. This site contains the current guidelines and forthcoming standards for Web accessibility.

deafness.about.com/health/deafness/msubmenu6.htm
This is the home page of **deafness.about.com**. It is a resource to find information pertaining to deafness.

www.cast.org
CAST (Center for Applied Special Technology) offers software, including a valuable accessibility checker, that help individuals with disabilities use a computer. The accessibility checker is a Web-based program that validates the accessibility of Web sites.

www.trainingpost.org/3-2-inst.htm
This site presents a tutorial on the Gunning Fog Index. The Gunning Fog Index grades text based on its readability.

www.w3.org/TR/REC-CSS2/aural.html
This page discusses Aural Style Sheets, outlining the purpose and uses of this new technology.

laurence.canlearn.ca/English/learn/newaccessguide/indie
INDIE stands for "Integrated Network of Disability Information and Education." This site is home to a search engine that helps users find information on disabilities.

java.sun.com/products/java-media/speech/forDevelopers/JSML
This site outlines the specifications for JSML, Sun Microsystem's Java Speech Markup Language. This language, like VoiceXML, could drastically improve accessibility for people with visual impairments.

www.slcc.edu/webguide/lynxit.html
Lynxit is a development tool that allows users to view any Web site as a text-only browser would. The site's form allows you to enter a URL and returns the Web site in text-only format.

www.trill-home.com/lynx/public_lynx.html
This site allows users to browse the Web with a Lynx browser. Users can view how Web pages appear to users without the most current technologies.

www.wgbh.org/wgbh/pages/ncam/accesslinks.html
This site provides links to other accessibility pages across the Web.

ocfo.ed.gov/coninfo/clibrary/software.htm
This page is the U.S. Department of Education's Web site for software accessibility requirements. It helps developers produce accessible products.

www-3.ibm.com/able/access.html
The homepage of IBM's accessibility site provides information on IBM products and their accessibility and discusses hardware, software and Web accessibility.

www.w3.org/TR/voice-tts-reqs
This page explains the speech synthesis markup requirements for voice markup languages.

www.voicexmlcentral.com
This site contains information about VoiceXML, such as the specification and the document type definition (DTD).

deafness.about.com/health/deafness/msubvib.htm
This site provides information on vibrotactile devices, which allow individuals with hearing impairments to experience audio in the form of vibrations.

web.ukonline.co.uk/ddmc/software.html
This site provides links to software for people with disabilities.

www.hj.com
Henter-Joyce is a division of Freedom Scientific that provides software for people with visual impairments. It is the home of JAWS.

www.abledata.com/text2/icg_hear.htm
This page contains a consumer guide that discusses technologies for people with hearing impairments.

www.washington.edu/doit
The University of Washington's DO-IT (Disabilities, Opportunities, Internetworking and Technology) site provides information and Web development resources for creating universally accessible Web sites.

www.webable.com
WebABLE contains links to many disability-related Internet resources and is geared towards those developing technologies for people with disabilities.

www.webaim.org
The *WebAIM* site provides a number of tutorials, articles, simulations and other useful resources that demonstrate how to design accessible Web sites. The site provides a screen reader simulation.

www.speech.cs.cmu.edu/comp.speech/SpeechLinks.html
The *Speech Technology Hyperlinks* page has over 500 links to sites related to computer-based speech and speech recognition tools.

www.islandnet.com/~tslemko
The *Micro Consulting Limited* site contains shareware speech synthesis software.

www.chantinc.com/technology
This page is the *Chant* Web site, which discusses speech technology and how it works. Chant also provides speech synthesis and speech recognition software.

whatis.techtarget.com/definition
This site provides definitions and information about several topics, including CallXML. Its thorough definition of CallXML differentiates CallXML and VoiceXML, another technology developed by Voxeo. The site contains links to other published articles discussing CallXML.

www.oasis-open.org/cover/callxmlv2.html
This site provides a comprehensive list of the CallXML tags complete with descriptions of each tag. Short examples on how to apply the tags in various applications are provided.

SUMMARY

- Enabling a Web site to meet the needs of individuals with disabilities is an issue relevant to all business owners.
- Legal ramifications exist for Web sites that discriminate against people with disabilities (i.e., by not providing them with adequate access to the site's resources).
- Technologies such as voice activation, visual enhancers and auditory aids enable individuals with disabilities to work in more positions.

- On April 7, 1997, the World Wide Web Consortium (W3C) launched the Web Accessibility Initiative (WAI). The WAI is an attempt to make the Web more accessible; its mission is described at **www.w3.org/WAI**.
- Accessibility refers to the level of usability of an application or Web site for people with disabilities. Total accessibility is difficult to achieve because there are many different disabilities, language barriers, and hardware and software inconsistencies.
- The majority of Web sites are considered either partially or totally inaccessible to people with visual, learning or mobility impairments.
- The WAI publishes the Web Content Accessibility Guidelines 1.0, which assigns priorities to a three-tier structure of checkpoints. The WAI currently is working on a draft of the Web Content Accessibility Guidelines 2.0.
- One important WAI requirement is to ensure that every image, movie and sound on a Web site is accompanied by a description that clearly defines the object's purpose; this is called an **<alt>** tag.
- Specialized user agents, such as screen readers (programs that allow users to hear what is being displayed on their screen) and braille displays (devices that receive data from screen-reading software and output the data as braille), allow people with visual impairments to access text-based information that is normally displayed on the screen.
- Using a screen reader to navigate a Web site can be time consuming and frustrating, because screen readers are unable to interpret pictures and other graphical content that do not have alternative text.
- Including links at the top of each Web page provides easy access to page's main content.
- Web pages with large amounts of multimedia content are difficult for user agents to interpret unless they are designed properly. Images, movies and most non-XHTML objects cannot be read by screen readers.
- Web designers should avoid misuse of the **alt** attribute; it is intended to provide a short description of an XHTML object that may not load properly on all user agents.
- The value of the **longdesc** attribute is a text-based URL, linked to a Web page, that describes the image associated with the attribute.
- When creating a Web page intended for the general public, it is important to consider the reading level at which it is written. Web site designers can make their sites more readable through the use of shorter words, as some users may have difficulty reading long words. In addition, users from other countries may have difficulty understanding slang and other nontraditional language.
- Web designers often use frames to display more than one XHTML file at a time and are a convenient way to ensure that certain content is always on screen. Unfortunately, frames often lack proper descriptions, which prevents users with text-based browsers, or users who lack sight, from navigating the Web site.
- The **<noframes>** tag allows the designer to offer alternative content to users whose browsers do not support frames.
- VoiceXML has tremendous implications for people with visual impairments and for the illiterate. VoiceXML, a speech recognition and synthesis technology, reads Web pages to users and understands words spoken into a microphone.
- A VoiceXML document is made up of a series of dialogs and subdialogs, which result in spoken interaction between the user and the computer. VoiceXML is a voice-recognition technology.
- CallXML, a language created and supported by Voxeo, creates phone-to-Web applications.

- When a user accesses a CallXML application, the incoming telephone call is referred to as a session. A CallXML application can support multiple sessions that enable the application to receive multiple telephone calls at any given time.
- A session terminates either when the user hangs up the telephone or when the CallXML application invokes the **hangup** element.
- The contents of a CallXML application are inserted within the **<callxml>** tag.
- CallXML tags that perform similar tasks should be enclosed within the **<block>** and **</block>** tags.
- To deploy a CallXML application, register with the Voxeo Community, which assigns a telephone number to the application so that other users may access it.
- Voxeo's logging feature enables developers to debug their telephone application by observing the "conversation" between the user and the application.
- Braille keyboards are similar to standard keyboards, except that in addition to having each key labeled with the letter it represents, braille keyboards have the equivalent braille symbol printed on the key. Most often, braille keyboards are combined with a speech synthesizer or a braille display, so users can interact with the computer to verify that their typing is correct.
- People with visual impairments are not the only beneficiaries of the effort to improve markup languages. Individuals with hearing impairments also have a great number of tools to help them interpret auditory information delivered over the Web.
- Speech synthesis is another research area that helps people with disabilities.
- Open-source software for people with visual impairments already exists and is often superior to most of its proprietary, closed-source counterparts.
- People with hearing impairments benefit from Synchronized Multimedia Integration Language (SMIL). This markup language adds extra tracks—layers of content found within a single audio or video file. The additional tracks can contain data such as closed captioning.
- EagleEyes, developed by researchers at Boston College (**www.bc.edu/eagleeyes**), translates eye movements into mouse movements. Users move the mouse cursor by moving their eyes or heads and are thereby are able to control computers.
- All of the accessibility options provided by Windows 2000 are available through the **Accessibility Wizard**. The **Accessibility Wizard** takes a user step by step through all of the Windows accessibility features and configures his or her computer according to the chosen specifications.
- Microsoft Magnifier enlarges the section of your screen surrounding the mouse cursor.
- To solve problems seeing the mouse cursor, Microsoft offers the ability to use larger cursors, black cursors and cursors that invert objects underneath them.
- **SoundSentry** is a tool that creates visual signals when system events occur.
- **ShowSounds** adds captions to spoken text and other sounds produced by today's multimedia-rich software.
- **StickyKeys** is a program that helps users who have difficulty pressing multiple keys at the same time.
- **BounceKeys** forces the computer to ignore repeated keystrokes, solving the problem of accidentally pressing the same key more than once.
- **ToggleKeys** causes an audible beep to alert users that they have pressed one of the lock keys (i.e., *Caps Lock*, *Num Lock*, or *Scroll Lock*).
- **MouseKeys** is a tool that uses the keyboard to emulate mouse movements.

- The **Mouse Button Settings** tool allows you to create a virtual left-handed mouse by swapping the button functions.
- A timeout either enables or disables a certain action after the computer has idled for a specified amount of time. A common example of a timeout is a screen saver.
- You can create an **`.acw`** file, that, when clicked, will automatically activate the saved accessibility settings on any Windows 2000 computer.
- Microsoft **Narrator** is a text-to-speech program for people with visual impairments. It reads text, describes the current desktop environment and alerts the user when certain Windows events occur.

TERMINOLOGY

accessibility
Accessibility Wizard
Accessibility Wizard: Display Color Settings
Accessibility Wizard: Icon Size
Accessibility Wizard: Mouse Cursor
Accessibility Wizard: Scroll Bar and Window Border Size
action element
alt attribute
Americans with Disabilities Act (ADA)
<assign> tag in VoiceXML
AuralCSS
<block> tag in VoiceXML
BounceKeys
braille display
braille keyboard
<break> tag in VoiceXML
<b> tag (bold)
CallXML
<callxml> tag in CallXML
caption
Cascading Style Sheets (CSS)
count attribute in VoiceXML
<choice> tag in VoiceXML
CSS2
D-link
default setting
EagleEyes
encoding
<enumerate> tag in VoiceXML
event handler
<exit> tag in VoiceXML
field variable
<filled> tag in VoiceXML
<form> tag in VoiceXML
frames
get request type
<getDigits> tag in CallXML
global variable
<goto> tag in VoiceXML
<grammar> tag in VoiceXML
Gunning Fog Index
header cells
headers attribute
<h1> tag
<strong> tag
style sheet
system carat
<subdialog> tag in VoiceXML
summary attribute
Synchronized Multimedia Integration Language (SMIL)
tables
<td> tag
termDigits attribute in CallXML
<text> tag in CallXML
text-to-speech (TTS)
<th> tag
IBM ViaVoice
id attribute
<img> tag
JAWS (Job Access With Sound)
level attribute in VoiceXML
linearize
<link> tag in VoiceXML
local dialog
logging feature
logic element
longdesc attribute
Lynx
markup language
maxDigits attribute in CallXML
maxTime attribute in CallXML
<menu> tag in VoiceXML
Microsoft Magnifier
Microsoft Narrator
Mouse Button Settings window
MouseKeys
Narrator
<next> tag in VoiceXML
nolimit (default value)
<noframes> tag
Ocularis
<onHangup> tag in CallXML
<onMaxSilence> tag in CallXML
On-Screen Keyboard
<onTermDigits> tag in CallXML
post request type
priority 1 checkpoint
priority 2 checkpoint
priority 3 checkpoint
<prompt> tag in VoiceXML
quick tip
readability
Read typed characters
screen reader
session
sessionID
Set Automatic Timeout window
ShowSounds
SoundSentry
speech recognition
speech synthesizer
StickyKeys
track
Unicode
user agent
<var> tag in VoiceXML
var attribute in CallXML
version
ViaVoice
voice server
Voice Server SDK
VoiceXML
Voxeo Community
<vxml> tag in VoiceXML
Web Accessibility Initiative (WAI)

timeout
<title> tag
ToggleKeys
Web Content Accessibility Guidelines 1.0
XML declaration
XML Guidelines (XML GL)

SELF-REVIEW EXERCISES

25.1 Expand the following acronyms:
a) W3C.
b) WAI.
c) JAWS.
d) SMIL.
e) CSS.

25.2 Fill in the blanks in each of the following statements.
a) The highest priority of the Web Accessibility Initiative ensures that each __________, __________ and __________ is accompanied by a description that clearly defines its purpose.
b) Technologies such as __________, __________ and __________ enable individuals with disabilities to work in a large number of positions.
c) Although they can be used as a great layout tool, __________ are difficult for screen readers to interpret and convey clearly to a user.
d) To make a frame accessible to individuals with disabilities, it is important to include __________ tags on a Web page.
e) __________ and __________ often assist blind people using computers.
k) CallXML creates __________ applications that allow businesses to receive and send telephone calls.
l) A __________ tag must be associated with the **<getDigits>** tag.

25.3 State whether each of the following is *true* or *false*. If *false*, explain why.
a) Screen readers have no problem reading and translating images.
b) When writing pages for the general public, it is important to consider the reading difficulty level of the text.
c) The **<alt>** tag helps screen readers describe images in a Web page.
d) Left-handed people have been helped by the improvements made in speech-recognition technology more than any other group of people.
e) VoiceXML lets users interact with Web content using speech recognition and speech synthesis technologies.
f) Elements such as **onMaxSilence** and **onTermDigit** are event handlers because they perform a specified task when invoked.
g) The debugging feature of the **Voxeo Account Manager** assists developers in debugging their CallXML application.

ANSWERS TO SELF-REVIEW EXERCISES

25.1 a) World Wide Web Consortium. b) Web Accessibility Initiative. c) Job Access with Sound. d) Synchronized Multimedia Integration Language. e) Cascading Style Sheets.

25.2 a) image, movie, sound. b) voice activation, visual enhancers and auditory aids. c) tables. d) **<noframes>**. e) Braille displays, braille keyboards. f) phone-to-Web. g) **<onTermDigit>**.

25.3 a) False. Screen readers have no way of telling a user what is shown in an image. If the programmer includes an **alt** attribute inside the **<img>** tag, the screen reader reads this description to the user. b) True. c) True. d) False. Although left-handed people can use speech-recognition technol-

ogy as everyone else can, speech-recognition technology has had the largest impact on the blind and on people who have trouble typing. e) True. f) True. g) False. The logging feature assists developers in debugging their CallXML application.

EXERCISES

25.4 Insert XHTML markup into each segment to make the segment accessible to someone with disabilities. The contents of images and frames should be apparent from the context and filenames.

a)
```
<img src = "dogs.jpg" width = "300" height = "250" />
```
b)
```
<table width = "75%">
   <tr><th>Language</th><th>Version</th></tr>
   <tr><td>XHTML</td><td>1.0</td></tr>
   <tr><td>Perl</td><td>5.6.0</td></tr>
   <tr><td>Java</td><td>1.3</td></tr>
</table>
```

25.5 Define the following terms:

a) Action element.
b) Gunning Fog Index.
c) Screen reader.
c) Session.
d) Web Accessibility Initiative (WAI).

25.6 Describe the three-tier structure of checkpoints (priority-one, priority-two and priority-three) set forth by the WAI.

25.7 Why do misused `<h1>` heading tags create problems for screen readers?

25.8 Use CallXML to create a voice mail system that plays a voice mail greeting and records the message. Have friends and classmates call your application and leave a message.

(*** EXERCISE SOLUTIONS***)

EXERCISES

25.4 Insert XHTML markup into each segment to make the segment accessible to someone with disabilities. The contents of images and frames should be apparent from the context and filenames.

```
a) <img src = "dogs.jpg" width = "300" height = "250" />
b) <table width = "75%">
      <tr><th>Language</th><th>Version</th></tr>
      <tr><td>XHTML</td><td>1.0</td></tr>
      <tr><td>Perl</td><td>5.6.0</td></tr>
      <tr><td>Java</td><td>1.3</td></tr>
   </table>
c) <map name = "links">
      <area href = "index.html" shape = "rect"
         coords = "50, 120, 80, 150" />
      <area href = "catalog.html" shape = "circle"
         coords = "220, 30" />
   </map>
   <img src = "antlinks.gif" width = "300" height = "200"
      usemap = "#links" />
```

ANS: a)
```
<img src = "dogs.jpg" width = "300" height = "250"
   alt = "Image of Dogs" />
```
b)
```
<table width = "75%" summary = "This table provides the ver-
sion number of XHTML, Perl and Java used in this book.">
   <tr><th id = "Language">Language</th><th id =
         "Version">Version</th></tr>
   <tr><td headers = "Language">XHTML</td><td headers =
         "Version">1.0</td></tr>
   <tr><td headers = "Language">Perl</td><td headers =
         "Version">5.6.0</td></tr>
   <tr><td headers = "Language">Java</td><td headers =
         "Version">1.3</td></tr>
</table>
```
c)
```
<map name = "links">
   <area href = "index.html" shape = "rect"
      coords = "50, 120, 80, 150" alt = "Index" />
   <area href = "catalog.html" shape = "circle"
      coords = "220, 30" alt = "Catalog" />
</map>
<img src = "antlinks.gif" width = "300" height = "200"
   usemap = "#links" alt = "Links" />
```

25.5 Define the following terms:
a) Action element.
b) Gunning Fog Index.
c) Screen reader.
d) Session.
e) Web Accessibility Initiative (WAI).

ANS: a) Action elements perform specified tasks, such as answering an incoming telephone call or hanging up on a telephone call, during the current session. b) The Gunning Fog Index is a formula that produces a readability grade when applied to a text sample. It allows Web developers to evaluate the readability grade of their Web sites, so that the Web sites are accessible to the majority of users. c) A screen reader is a program that allows users to hear what is being displayed on their screen. d) A session is any given telephone call to which the CallXML application responds. A CallXML application can support multiple sessions at once, and each session is assigned a unique sessionID. e) The Web Accessibility Initiative (WAI) is an attempt to make the Web more accessible. The WAI published the Web Content Accessibility Guidelines (WCAG) 1.0 to set forth checkpoints that helped businesses determine if their Web sites are accessible to everyone.

25.6 Describe the three-tier structure of checkpoints (priority one, priority two and priority three) set forth by the WAI.

ANS: Priority-one checkpoints are goals that must be met in order to ensure accessibility. Priority-two checkpoints, though not essential, are highly recommended. Priority-three checkpoints slightly improve accessibility.

25.7 Why do `<h1>` heading tags that are inappropriately used create problems for screen readers?

ANS: The `<h1>` heading tags are often erroneously used to make text large and bold. The desired visual effect may be achieved, but it creates a problem for screen readers. When a screen reader software encounters `<h1>` tags, it may verbally inform the user that a new section has been reached, which may confuse the user. It is best to use `<h1>` tags in accordance with their XHTML specification (e.g., as headings to introduce important sections of a document).

25.8 Use CallXML to create a voice mail system that plays a voice mail greeting and records the message. Have friends and classmates call your application and leave a message.

ANS:

```
<?xml version = "1.0" encoding = "UTF-8"?>

<!-- Exercise 5: isbn_voice.xml   -->
<!-- Records a voice mail message -->

<callxml>
   <block>
      <text>
         Hi, you have reached Kate Steinbuhler. I cannot come to
         the phone right now, so please leave a message and I will
         return your call as soon as possible. To end the call,
        either press the pound sign, or simply hang up. Thank you.
      </text>

      <!-- Records a message spoken by the caller and sends -->
      <!-- the file to an email account. The user has 60    -->
      <!-- seconds to leave a message.                      -->
      <recordAudio format = "audio/wav"
         value = "mailto:rashmi.jayaprakash@deitel.net"
         termDigits = "#"
         maxDigits = "1"
         maxTime = "60s" />

      <!-- Requests that the user enter a message or end the -->
      <!-- call after the elapsed time of 60 seconds.         -->
      <onMaxSilence>
         <text>
            Please leave a message or press the pound sign to exit.
         </text>

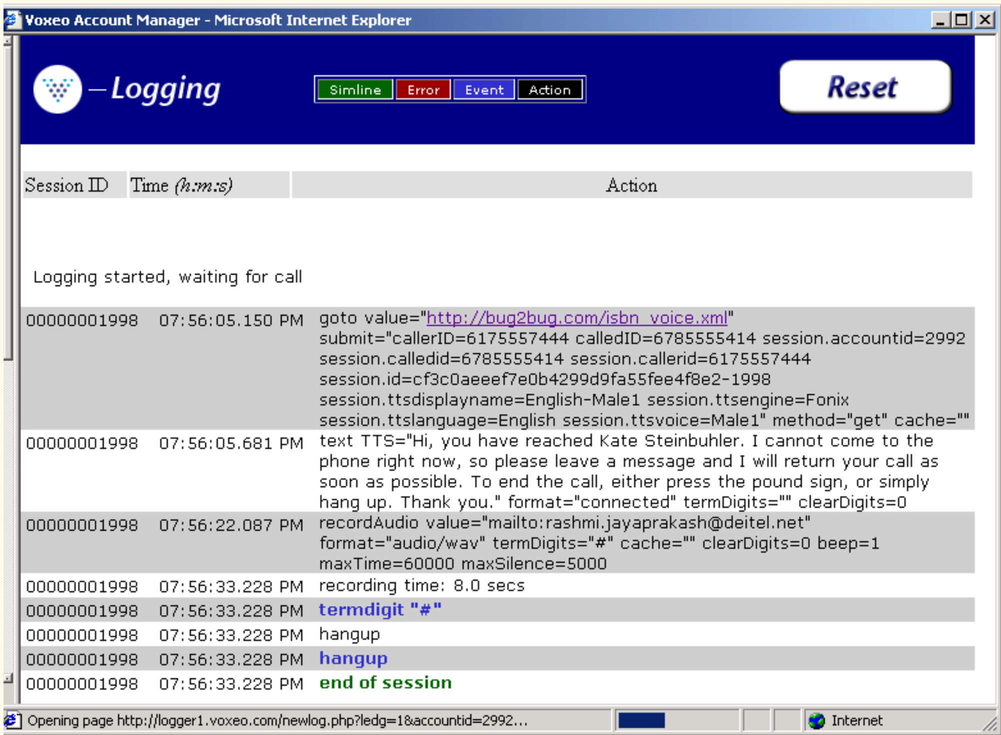
         <recordAudio format = "audio/wav"
            value = "mailto:rashmi.jayaprakash@deitel.net"
            termDigits = "#"
            maxDigits = "1"
            maxTime = "60s" />

      </onMaxSilence>

      <onTermDigit value = "#">
         <hangup/>
      </onTermDigit>
   </block>

   <!-- Event handler that terminates the call -->
   <onHangup />
</callxml>
```

CallXML example that acts as a voice mail system (part 1 of 2).



CallXML example that acts as a voice mail system (part 2 of 2).

[***Notes To Reviewers***]

- Please list URLs that discuss Python-specific accessibility issues. We are conducting our own research and will post this chapter for second round reviews after the inclusion of Python-specific material.
- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send us e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **cheryl.yaeger@deitel.net**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
- Please be constructive. This book will be published soon. We all want to publish the best possible book.
- If you find something that is incorrect, please show us how to correct it.
- Please read all the back matter including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

26

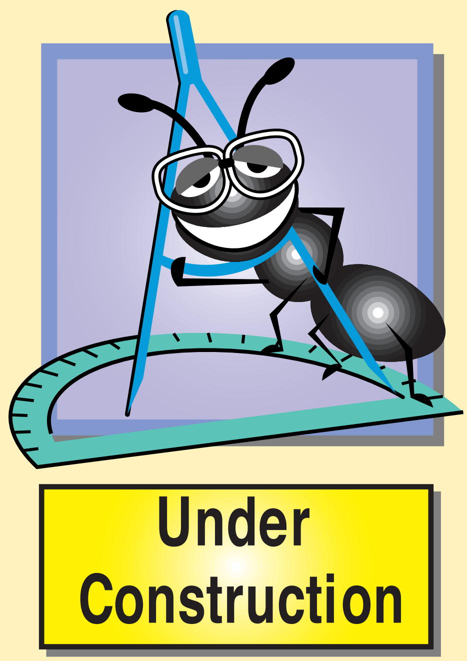
Introduction to XHTML: Part 1

Objectives

- To understand important components of XHTML documents.
- To use XHTML to create World Wide Web pages.
- To add images to Web pages.
- To understand how to create and use hyperlinks to navigate Web pages.
- To mark up lists of information.

To read between the lines was easier than to follow the text.
Henry James

High thoughts must have high language.
Aristophanes



Outline

26.1 Introduction

Welcome to the world of opportunity created by the World Wide Web. The Internet is now three decades old, but it was not until the World Wide Web became popular in the 1990s that the explosion of opportunity that we are still experiencing began. Exciting new developments occur almost daily—the pace of innovation is unprecedented by any other technology. In this chapter, you will develop your own Web pages. As the book proceeds, you will create increasingly appealing and powerful Web pages. In the later portion of the book, you will learn how to create complete Web-based applications.

In this chapter, we begin unlocking the power of Web-based application development with *XHTML*[1]—the *Extensible Hypertext Markup Language*. In later chapters, we introduce more sophisticated XHTML techniques, such as *tables*, which are particularly useful for structuring information from *databases* (i.e., software that stores structured sets of data), and *Cascading Style Sheets (CSS)*, which make Web pages more visually appealing.

Unlike procedural programming languages such as C, Fortran, Cobol and Pascal, XHTML is a *markup language* that specifies the format of text that is displayed in a Web browser such as Microsoft's Internet Explorer or Netscape's Communicator.

One key issue when using XHTML[2] is the separation of the *presentation of a document* (i.e., the document's appearance when rendered by a browser) from the *structure of the document's information*. Over the next several chapters, we discuss this issue in depth.

1. XHTML has replaced the HyperText Markup Language (HTML) as the primary means of describing Web content. XHTML provides more robust, richer and extensible features than HTML. For more on XHTML/HTML visit **www.w3.org/markup**.

26.2 Editing XHTML

In this chapter, we write XHTML in its *source-code form*. We create *XHTML documents* by typing them in with a text editor (e.g., *Notepad*, *Wordpad*, *vi*, *emacs*, etc.), saving the documents with either an ***.html*** or ***.htm*** file-name extension.

Good Programming Practice 26.1

Assign documents file names that describe their functionality. This practice can help you identify documents faster. It also helps people who want to link to a page, by giving them an easy-to-remember name. For example, if you are writing an XHTML document that contains product information, you might want to call it ***products.html****.*

Machines running specialized software called *Web servers* store XHTML documents. Clients (e.g., Web browsers) request specific *resources* from the Web server. For example, typing **www.deitel.com/books/downloads.htm** into a Web browser's address field requests **downloads.htm** from the Web server running at **www.deitel.com**. This document resides in a directory named **books**. For now, we simply place the XHTML documents on our machine and open them using Internet Explorer.

26.3 First XHTML Example[3]

In this chapter and the next, we present XHTML markup and provide screen captures that show how Internet Explorer 5.5 renders (i.e., displays) the XHTML. Every XHTML document we show has line numbers for the reader's convenience. These line numbers are not part of the XHTML documents.

Our first example (Fig. 26.1) is an XHTML document named **main.html** that displays the message "Welcome to XHTML!" in the browser.

The key line in the program is line 14, which tells the browser to display "Welcome to XHTML!" Now let us consider each line of the program.

Lines 1 is the optional *XML declaration* that identifies the **version** of XML used in the document. Line 2–3 is required for XHTML documents to conform with proper XHTML syntax.

Lines 5–6 are *XHTML comments*. Comments do not cause the browser to perform any action when the user loads the XHTML document into the Web browser to view the document. XHTML comments always start with **<!--** and end with **-->**. Each of our XHTML examples include comments that specify the figure number and file name, and provide a brief description of the example's purpose. Subsequent examples include comments in the markup, especially to highlight new features.

---

2. As this book was being submitted to the publisher, XHTML 1.1 became a World Wide Web Consortium (W3C) Recommendation. The XHTML examples presented in this book are based upon the XHTML 1.0 Recommendation, because Internet Explorer 5.5 does not support the full set of XHTML 1.1 features. In the future, Internet Explorer and other browsers will support XHTML 1.1. When this occurs, we will update our Web site (**www.deitel.com**) with XHTML 1.1 examples and information.
3. All of the examples presented in this book are available at **www.deitel.com** and on the CD-ROM that accompanies this book.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4  
5  <!-- Fig. 26.1: main.html -->
6  <!-- Our first Web page   -->
7  
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Welcome</title>
11    </head>
12 
13    <body>
14       <p>Welcome to XHTML!</p>
15    </body>
16 </html>
```

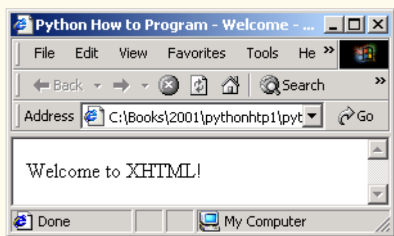


Fig. 26.1 First XHTML example.

XHTML markup contains text that represents the content of a document and *elements* that specify a document's structure. Some important elements of an XHTML document include the ***html*** element, the ***head*** element and the ***body*** element. The **html** element encloses the *head section* (represented by the ***head*** *element*) and the *body section* (represented by the ***body*** *element*). The head section contains information about the XHTML document, such as the *title* of the document. The head section also can contain special document formatting instructions called *style sheets* and client-side programs called *scripts* for creating dynamic Web pages. (We introduce style sheets in Chapter 28.) The body section contains the page's content that the browser displays when the user visits the Web page.

XHTML documents delimit an element with *start* and *end* tags. A start tag consists of the element name in angle brackets (e.g., **<html>**). An end tag consists of the element name preceded by a **/** in angle brackets (e.g., **</html>**). In this example lines 8 and 16 define the start and end of the **html** element. Note that the end tag on line 16 has the same name as the start tag, but is preceded by a **/** inside the angle brackets. Many start tags define *attributes* that provide additional information about an element. Browsers can use this additional information to determine how to process the element. Each attribute has a *name* and a *value* separated by an equal sign (**=**). Line 8 specifies a required attribute (**xmlns**) and value (**http://www.w3.org/1999/xhtml**) for the **html** element in an XHTML document. Simply copy and paste the **html** element start tag on line 8 into your XHTML documents.

Common Programming Error 26.1

Not enclosing attribute values in either single or double quotes is a syntax error.

Common Programming Error 26.2

Using uppercase letters in an XHTML element or attribute name is a syntax error.

An XHTML document divides the **html** element into two sections—head and body. Lines 9–11 define the Web page's head section with a **head** element. Line 10 specifies a **title** element. This is called a *nested element*, because it is enclosed in the **head** element's start and end tags. The **head** element also is a nested element, because it is enclosed in the **html** element's start and end tags. The **title** element describes the Web page. Titles usually appear in the *title bar* at the top of the browser window and also as the text identifying a page when users add the page to their list of **Favorites** or **Bookmarks**, which enable users to return to their favorite sites. Search engines (i.e., sites that allow users to search the Web) also use the **title** for cataloging purposes.

Good Programming Practice 26.2

Indenting nested elements emphasizes a document's structure and promotes readability.

Common Programming Error 26.3

XHTML does not permit tags to overlap—a nested element's end tag must appear in the document before the enclosing element's end tag. For example, the nested XHTML tags ***<head><title>hello</head></title>*** *cause a syntax error, because the enclosing* ***head*** *element's ending* ***</head>*** *tag appears before the nested* ***title*** *element's ending* ***</title>*** *tag.*

Good Programming Practice 26.3

Use a consistent ***title*** *naming convention for all pages on a site. For example, if a site is named "Bailey's Web Site," then the* ***title*** *of the main page might be "Bailey's Web Site—Links," etc. This practice can help users better understand the Web site's structure.*

Line 13 opens the document's ***body*** element. The body section of an XHTML document specifies the document's content, which may include text and tags.

Some tags, such as the *paragraph tags* (**<p>** and **</p>**) in line 14, markup text for display in a browser. All text placed between the **<p>** and **</p>** tags form one paragraph. When the browser renders a paragraph, a blank line usually precedes and follows paragraph text.

This document ends with two closing tags (lines 15–16). These tags close the **body** and **html** elements, respectively. The ending **</html>** tag in an XHTML document informs the browser that the XHTML markup is complete.

To view this example in Internet Explorer, perform the following steps:

1. Copy the Chapter 26 examples onto your machine from the CD that accompanies this book (or download the examples from **www.deitel.com**).
2. Launch Internet Explorer and select **Open...** from the **File** Menu. This displays the **Open** dialog.

3. Click the **Open** dialog's **Browse...** button to display the **Microsoft Internet Explorer** file dialog.
4. Navigate to the directory containing the Chapter 26 examples and select the file **`main.html`**, then click **Open**.
5. Click **OK** to have Internet Explorer render the document. Other examples are opened in a similar manner.

At this point your browser window should appear similar to the sample screen capture shown in Fig. 26.1. (Note that we resized the browser window to save space in the book.)

26.4 W3C XHTML Validation Service

Programming Web-based applications can be complex and XHTML documents must be written correctly to ensure that browsers process them properly. To promote correctly written documents, the *World Wide Web Consortium (W3C)* provides a *validation service* (**`validator.w3.org`**) for checking a document's syntax. Documents can be validated from either a URL that specifies the location of the file or by uploading a file to the site **`validator.w3.org/file-upload.html`**. Uploading a file copies the file from the user's computer to another computer on the Internet. Figure 26.2 shows **`main.html`** (Fig. 26.1) being uploaded for validation. Although the W3C's Web page indicates that the service name is **HTML Validation Service**,[4] the service validatse the syntax of XHTML documents. All the XHTML examples in this book are validated successfully using **`validator.w3.org`**.

4. HTML (HyperText Markup Language) is the predecessor of XHTML designed for marking up Web content. HTML is a deprecated technology.

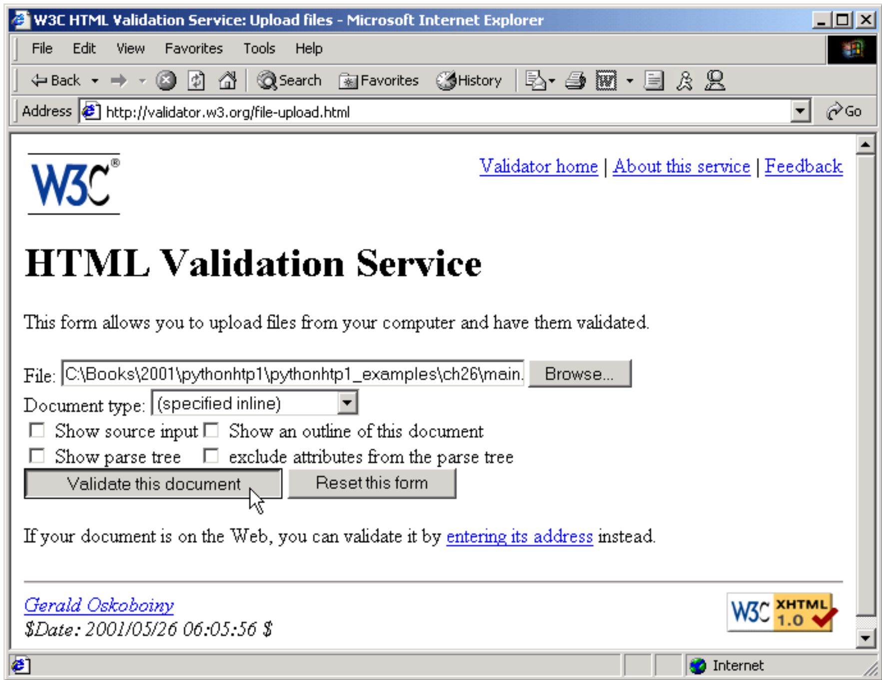


Fig. 26.2 Validating an XHTML document.

By clicking **Browse...**, users can select files on their own computers for upload. After selecting a file, clicking the **Validate this document** button uploads and validates the file. Figure 26.3 shows the results of validating **`main.html`**. This document does not contain any syntax errors. If a document does contain syntax errors, the validation service displays error messages describing the errors. Figure 26.4 shows the results of validating the **`main.html`** document, which contains a syntax error—the closing **`</p>`** tag is omitted. In Exercise 26.13, we ask readers to create an invalid XHTML document (i.e., one that contains syntax errors) and to check the document's syntax using the validation service. This enables readers to see the types of error messages generated by the validator.

Testing and Debugging Tip 26.1

Use a validation service, such as the W3C HTML Validation Service, to confirm that an XHTML document is correct syntactically.

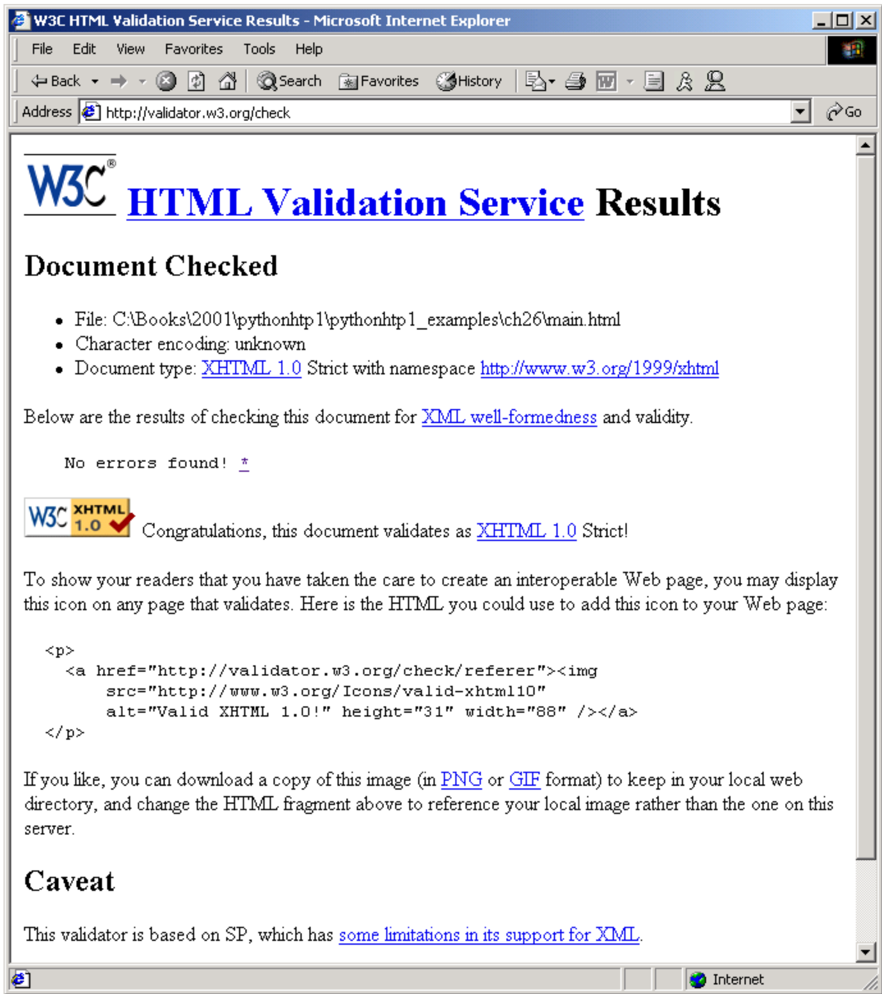


Fig. 26.3 XHTML validation results of **`main.html`** containing no syntax errors.

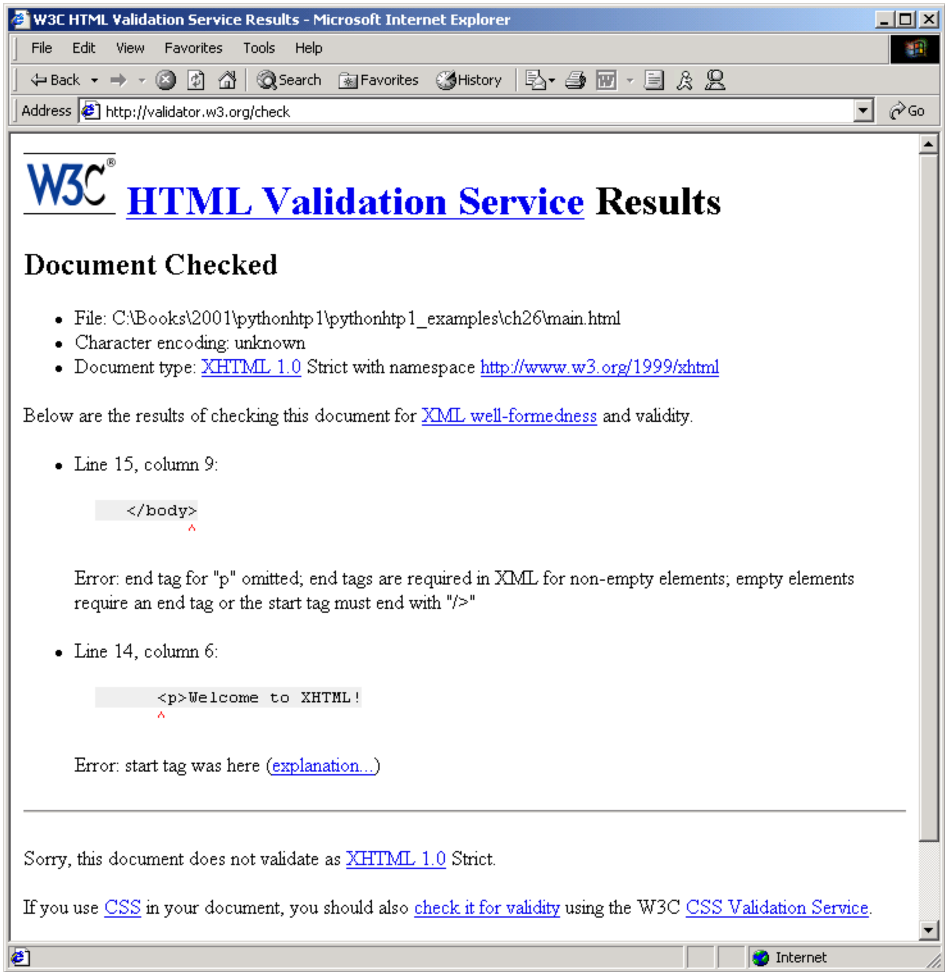


Fig. 26.4 XHTML validation results of **`main.html`** containing a syntax error.

26.5 Headers

Some text in an XHTML document may be more important than others. For example, the text in this section is considered more important than a footnote. XHTML provides six *headers*, called *header elements*, for specifying the relative importance of information. Figure 26.5 demonstrates these elements (**`h1`** through **`h6`**).

Portability Tip 26.1

The text size used to display each header element can vary significantly between browsers. In Chapter 28, we discuss how to control the text size and other text properties.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 26.5: header.html -->
6  <!-- XHTML headers         -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Headers</title>
11    </head>
12
13    <body>
14
15       <h1>Level 1 Header</h1>
16       <h2>Level 2 header</h2>
17       <h3>Level 3 header</h3>
18       <h4>Level 4 header</h4>
19       <h5>Level 5 header</h5>
20       <h6>Level 6 header</h6>
21
22    </body>
23 </html>
```

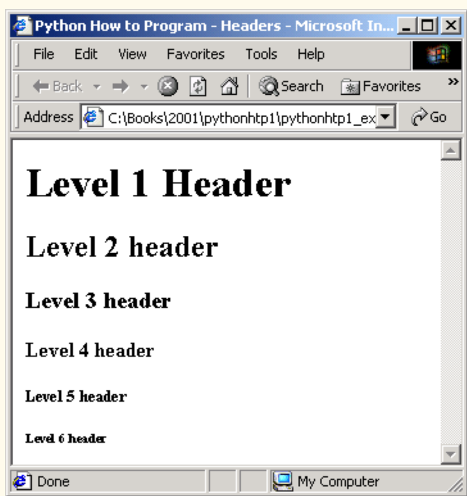


Fig. 26.5 Header elements **h1** through **h6**.

Header element **h1** (line 15) is considered the most significant header and is rendered in a larger font than the other five headers (lines 16–20). Each successive header element (i.e., **h2**, **h3**, etc.) is rendered in a smaller font.

Look-and-Feel Observation 26.1

Placing a header at the top of every XHTML page helps viewers understand the purpose of each page.

Look-and-Feel Observation 26.2

Use larger headers to emphasize more important sections of a Web page.

26.6 Linking

One of the most important XHTML features is the *hyperlink,* which references (or *links* to) other resources such as XHTML documents and images. In XHTML, both text and images can act as hyperlinks. Web browsers typically underline text hyperlinks and color their text blue by default, so that users can distinguish hyperlinks from plain text. In Fig. 26.6, we create text hyperlinks to four different Web sites.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 26.6: links.html      -->
<!-- Introduction to hyperlinks -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Links</title>
   </head>

   <body>

      <h1>Here are my favorite sites</h1>

      <p><strong>Click a name to go to that page.</strong></p>

      <!-- Create four text hyperlinks -->
      <p><a href = "http://www.deitel.com">Deitel</a></p>

      <p><a href = "http://www.prenhall.com">Prentice Hall</a></p>

      <p><a href = "http://www.yahoo.com">Yahoo!</a></p>

      <p><a href = "http://www.usatoday.com">USA Today</a></p>

   </body>
</html>
```

Fig. 26.6 Linking to other Web pages (part 1 of 2).

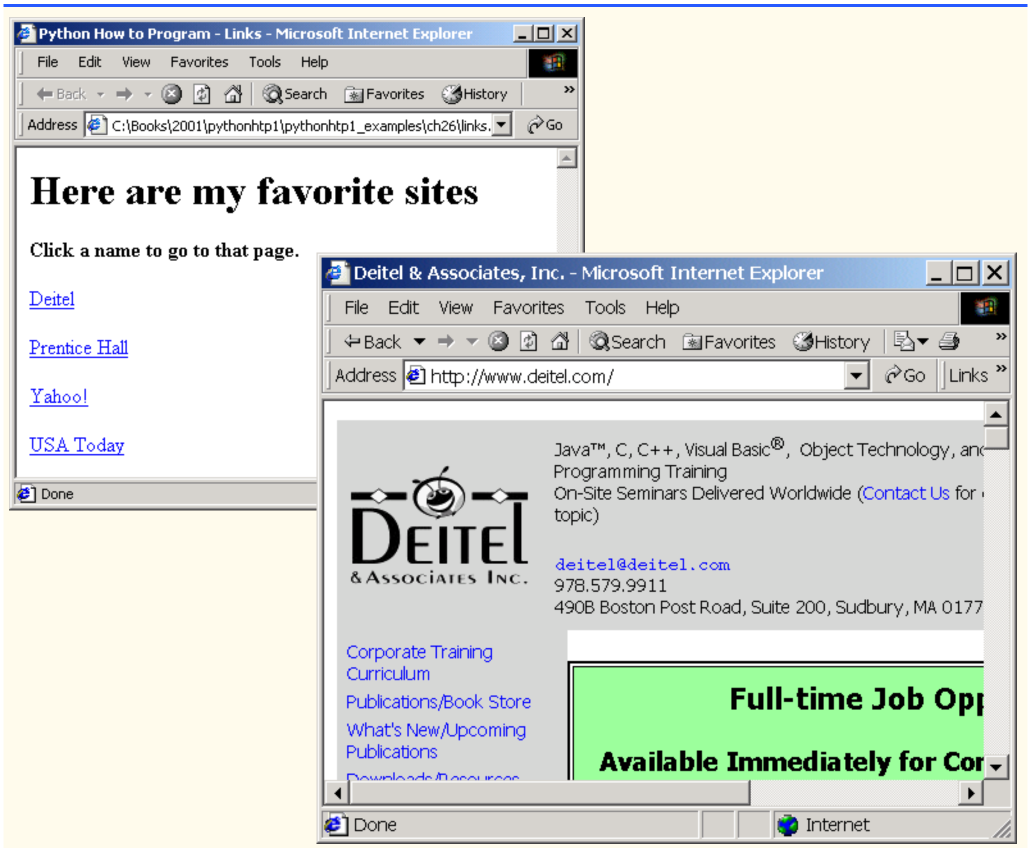


Fig. 26.6 Linking to other Web pages (part 2 of 2).

Line 17 introduces the **`<strong>`** tag. Browsers typically display text marked up with **`<strong>`** in a bold font.

Links are created using the ***`a`*** (*anchor*) *element*. Line 20 defines a hyperlink that links the text **`Deitel`** to the URL assigned to attribute ***`href`***, which specifies the location of a linked resource, such as a Web page, a file or an e-mail address. This particular anchor element links to a Web page located at **`http://www.deitel.com`**. When a URL does not indicate a specific document on the Web site, the Web server returns a default Web page. This pages often is called **`index.html`**; however, most Web servers can be configured to to use any file as the default Web page for the site. (Open **`http://www.deitel.com`** in one browser window and **`http://www.deitel.com/index.html`** in a second browser window to confirm that they are identical.) If the Web server cannot locate a requested document, the server returns an error indication to the Web browser and the browser displays an error message to the user.

Anchors can link to e-mail addresses using a ***`mailto:`*** URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Outlook Express) to enable the user to write an e-mail message to the linked address. Figure 26.7 demonstrates this type of anchor.

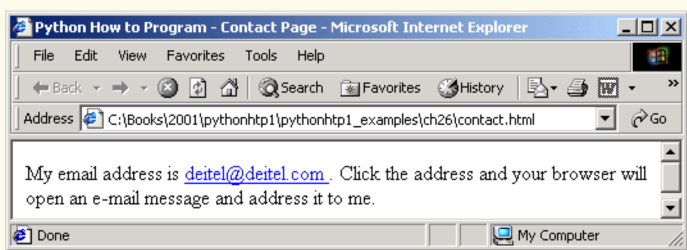
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 26.7: contact.html -->
<!-- Adding email hyperlinks -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Contact Page
      </title>
   </head>

   <body>

      <p>My email address is
         <a href = "mailto:deitel@deitel.com">
            deitel@deitel.com
         </a>
         . Click the address and your browser will
         open an e-mail message and address it to me.
      </p>
   </body>
</html>
```



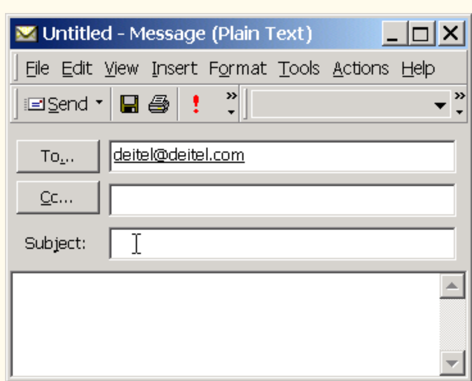


Fig. 26.7 Linking to an e-mail address.

Lines 17–19 contain an e-mail link. The form of an e-mail anchor is ****…****. In this case, we link to the e-mail address **deitel@deitel.com**.

26.7 Images

The examples discussed so far demonstrated how to mark up documents that contain only text. However, most Web pages contain both text and images. In fact, images are an equal, if not essential, part of Web-page design. The two most popular image formats used by Web developers are *Graphics Interchange Format (GIF)* and J*oint Photographic Experts Group (JPEG)* images. Users can create images using specialized pieces of software such as Adobe PhotoShop Elements and Jasc Paint Shop Pro[5]. Images may also be acquired from various Web sites, such as **gallery.yahoo.com**. Figure 26.8 demonstrates how to incorporate images into Web pages.

Lines 15–16 use an ***img*** element to insert an image in the document. The image file's location is specified with the **img** element's ***src*** attribute. In this case, the image is located in the same directory as this XHTML document, so only the image's file name is required. Optional attributes ***width*** and ***height*** specify the image's width and height, respectively. The document author can scale an image by increasing or decreasing the values of the image **width** and **height** attributes. If these attributes are omitted, the browser uses the image's actual width and height. Images are measured in *pixels* ("picture elements"), which represent dots of color on the screen. The image in Fig. 26.8 is **183** pixels wide and **238** pixels high.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 26.8: picture.html  -->
6  <!-- Adding images with XHTML -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Welcome</title>
11    </head>
12
13    <body>
14
15       <p><img src = "xmlhtp.jpg" height = "238" width = "183"
16             alt = "XML How to Program book cover" />
17          <img src = "jhtp.jpg" height = "238" width = "183"
18             alt = "Java How to Program book cover" />
19       </p>
20    </body>
21 </html>
```

Fig. 26.8 Placing images in XHTML files (part 1 of 2).

5. The CD-ROM that accompanies this book contains a 90-day evaluation version of Paint Shop Pro™.

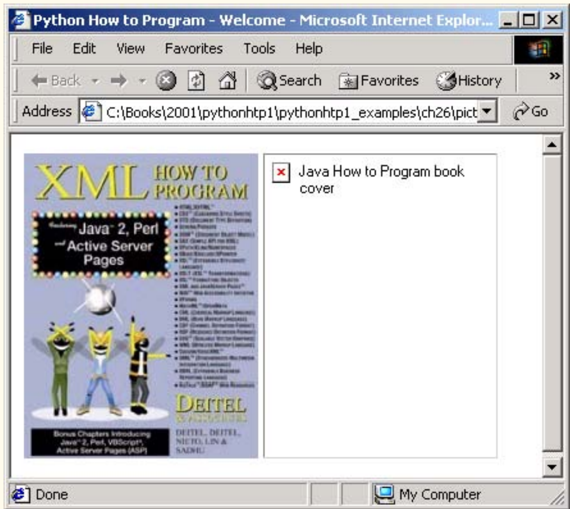


Fig. 26.8 Placing images in XHTML files (part 2 of 2).

Good Programming Practice 26.4

Always include the **`width`** *and the* **`height`** *of an image inside the* **`<img>`** *tag. When the browser loads the XHTML file, it knows immediately from these attributes how much screen space to provide for the image and lays out the page properly, even before it downloads the image.*

Performance Tip 26.1

Including the **`width`** *and* **`height`** *attributes in an* **`<img>`** *tag can result in the browser loading and rendering pages faster.*

Common Programming Error 26.4

Entering new dimensions for an image that change its inherent width-to-height ratio distorts the appearance of the image. For example, if your image is 200 pixels wide and 100 pixels high, you should ensure that any new dimensions have a 2:1 width-to-height ratio.

Every **`img`** element in an XHTML document has an ***`alt`*** attribute. If a browser cannot render an image, the browser displays the **`alt`** attribute's value. A browser may not be able to render an image for several reasons. It may not support images—as is the case with a *text-based browser* (i.e., a browser that can display only text)—or the client may have disabled image viewing to reduce download time. Figure 26.8 shows Internet Explorer 5.5 rendering the **`alt`** attribute's value when a document references a non-existent image file (**`jhtp.jpg`**).

Some XHTML elements (called *empty elements*) contain only attributes and do not markup text (i.e., text is not placed between the start and end tags). Empty elements (e.g., **`img`**) must be terminated, either by using the *forward slash character* (**`/`**) inside the closing right angle bracket (**`>`**) of the start tag or by explicitly including the end tag. When using the forward slash character, we add a space before the forward slash to improve readability

(as shown at the ends of lines 16 and 18). Rather than using the forward slash character, lines 17–18 could be written with a closing **</img>** tag as follows:

```
<img src = "jhtp.jpg" height = "238" width = "183"
   alt = "Java How to Program book cover"></img></p>
```

By using images as hyperlinks, Web developers can create graphical Web pages that link to other resources. In Fig. 26.9, we create six different image hyperlinks.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 26.9: nav.html          -->
<!-- Using images as link anchors -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Navigation Bar
      </title>
   </head>

   <body>

      <p>
         <a href = "links.html">
            <img src = "buttons/links.jpg" width = "65"
               height = "50" alt = "Links Page" />
         </a><br />

         <a href = "list.html">
            <img src = "buttons/list.jpg" width = "65"
               height = "50" alt = "List Example Page" />
         </a><br />

         <a href = "contact.html">
            <img src = "buttons/contact.jpg" width = "65"
               height = "50" alt = "Contact Page" />
         </a><br />

         <a href = "header.html">
            <img src = "buttons/header.jpg" width = "65"
               height = "50" alt = "Header Page" />
         </a><br />

         <a href = "table.html">
            <img src = "buttons/table.jpg" width = "65"
               height = "50" alt = "Table Page" />
         </a><br />

         <a href = "form.html">
            <img src = "buttons/form.jpg" width = "65"
```

Fig. 26.9 Using images as link anchors (part 1 of 2).

```
44                     height = "50" alt = "Feedback Form" />
45                </a><br />
46             </p>
47 
48          </body>
49       </html>
```

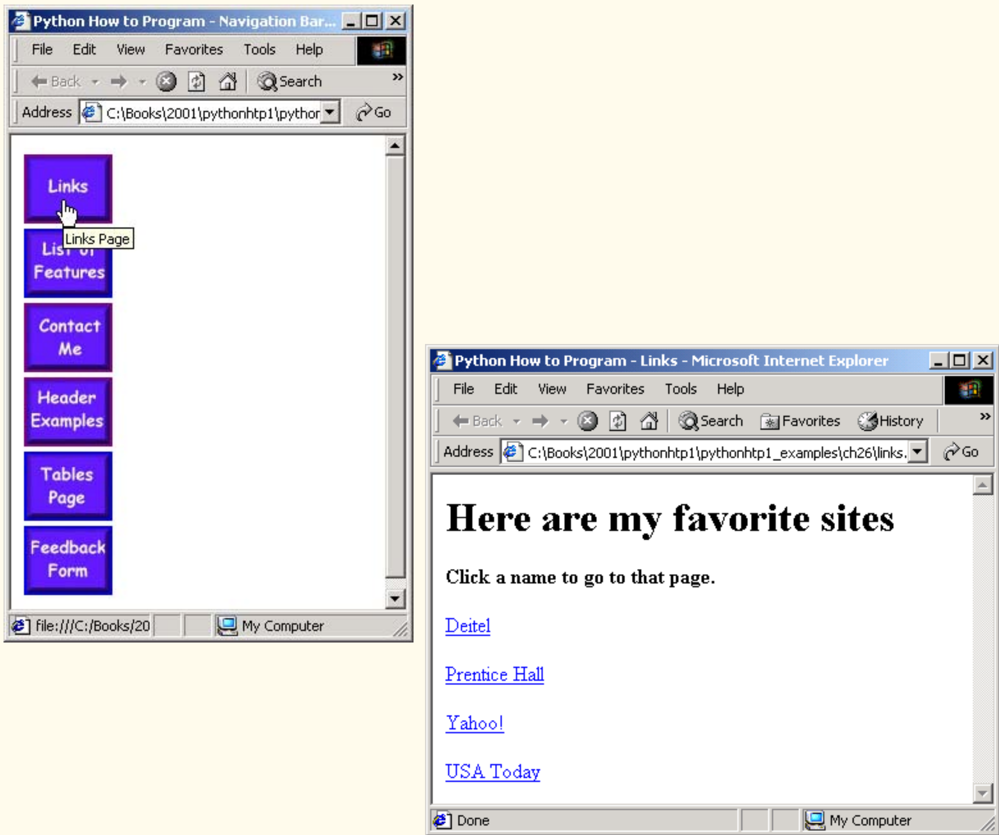


Fig. 26.9 Using images as link anchors (part 2 of 2).

Lines 17–20 create an *image hyperlink* by nesting an **img** element nested in an anchor (**a**) element. The value of the **img** element's **src** attribute value specifies that this image (**links.jpg**) resides in a directory named **buttons**. The **buttons** directory and the XHTML document are in the same directory. Images from other Web documents also can be referenced (after obtaining permission from the document's owner) by setting the **src** attribute to the name and location of the image.

On line 20, we introduce the ***br** element*, which most browsers render as a *line break*. Any markup or text following a **br** element is rendered on the next line. Like the **img** element, **br** is an example of an empty element terminated with a forward slash. We add a space before the forward slash to enhance readability.

26.8 Special Characters and More Line Breaks

When marking up text, certain characters or symbols (e.g., **<**) may be difficult to embed directly into an XHTML document. Some keyboards may not provide these symbols, or the presence of these symbols may cause syntax errors. For example, the markup

```
<p>if x < 10 then increment x by 1</p>
```

results in a syntax error because it uses the less-than character (**<**), which is reserved for start tags and end tags such as **<p>** and **</p>**. XHTML provides *special characters* or *entity references* (in the form **&**_code_**;**) for representing these characters. We could correct the previous line by writing

```
<p>if x &lt; 10 then increment x by 1</p>
```

which uses the special character ***<*** for the less-than symbol. Figure 26.10 demonstrates how to use special characters in an XHTML document.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 26.10: contact2.html    -->
<!-- Inserting special characters -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Contact Page
      </title>
   </head>

   <body>

      <!-- special characters are entered -->
      <!-- using the form &code;          -->
      <p>
         Click
         <a href = "mailto:deitel@deitel.com">here
         </a> to open an e-mail message addressed to
         deitel@deitel.com.
      </p>

      <hr /> <!-- inserts a horizontal rule -->

      <p>All information on this site is <strong>&copy;</strong>
         Deitel <strong>&amp;</strong> Associates, Inc. 2002.</p>

      <!-- to strike through text use <del> tags   -->
      <!-- to subscript text use <sub> tags        -->
      <!-- to superscript text use <sup> tags      -->
      <!-- these tags are nested inside other tags -->
```

Fig. 26.10 Inserting special characters into XHTML (part 1 of 2).

```
34         <p><del>You may download 3.14 x 10<sup>2</sup>
35            characters worth of information from this site.</del>
36            Only <sub>one</sub> download per hour is permitted.</p>
37
38         <p>Note: <strong>&lt; &frac14;</strong> of the information
39            presented here is updated daily.</p>
40
41      </body>
42   </html>
```

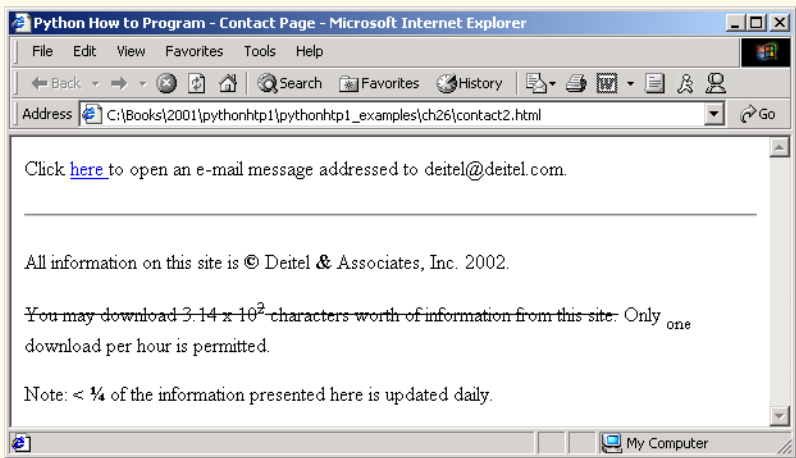


Fig. 26.10 Inserting special characters into XHTML (part 2 of 2).

Lines 27–28 contain other special characters, which are expressed as either word abbreviations (e.g., **amp** for ampersand and **copy** for copyright) or *hexadecimal (hex)* values (e.g., **&** is the hexadecimal representation of **&**). Hexadecimal numbers are base 16 numbers—digits in a hexadecimal number have values from 0 to 15 (a total of 16 different values). The letters A–F represent the hexadecimal digits corresponding to decimal values 10–15. Thus in hexadecimal notation we can have numbers like 876 consisting solely of decimal-like digits, numbers like DA19F consisting of digits and letters, and numbers like DCB consisting solely of letters. We discuss hexadecimal numbers in detail in Appendix C, Number Systems.

In lines 34–36, we introduce three new elements. Most browsers render the ***del*** element as strike-through text. With this format users can easily indicate document revisions. To *superscript* text (i.e., raise text on a line with a decreased font size) or *subscript* text (i.e., lower text on a line with a decreased font size), use the ***sup*** and ***sub*** elements, respectively. We also use special characters **<** for a less-than sign and ***¼*** for the fraction 1/4 (line 38).

In addition to special characters, this document introduces a *horizontal rule*, indicated by the ***<hr />*** tag in line 24. Most browsers render a horizontal rule as a horizontal line. The **<hr />** tag also inserts a line break above and below the horizontal line.

26.9 Unordered Lists

Up to this point, we have presented basic XHTML elements and attributes for linking to resources, creating headers, using special characters and incorporating images. In this section, we discuss how to organize information on a Web page using lists. In Chapter 27, we introduce another feature for organizing information, called a table. Figure 26.11 displays text in an *unordered list* (i.e., a list that does not order its items by letter or number). The *unordered list element* ***ul*** creates a list in which each item begins with a bullet symbol (called a *disc*).

Each entry in an unordered list (element ***ul*** in line 20) is an ***li*** (*list item*) element (lines 23, 25, 27 and 29). Most Web browsers render these elements with a line break and a bullet symbol indented from the beginning of the new line.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 26.11: links2.html               -->
6  <!-- Unordered list containing hyperlinks -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Links</title>
11    </head>
12
13    <body>
14
15       <h1>Here are my favorite sites</h1>
16
17       <p><strong>Click on a name to go to that page.</strong></p>
18
19       <!-- create an unordered list -->
20       <ul>
21
22          <!-- add four list items -->
23          <li><a href = "http://www.deitel.com">Deitel</a></li>
24
25          <li><a href = "http://www.w3.org">W3C</a></li>
26
27          <li><a href = "http://www.yahoo.com">Yahoo!</a></li>
28
29          <li><a href = "http://www.cnn.com">CNN</a></li>
30        </ul>
31    </body>
32 </html>
```

Fig. 26.11 Unordered lists in XHTML (part 1 of 2).



Fig. 26.11 Unordered lists in XHTML (part 2 of 2).

26.10 Nested and Ordered Lists

Lists may be nested to represent hierarchical relationships, as in an outline format. Figure 26.12 demonstrates nested lists and *ordered lists* (i.e., list that order their items by letter or number).

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 26.12: list.html              -->
<!-- Advanced Lists: nested and ordered -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Lists</title>
   </head>

   <body>

      <h1>The Best Features of the Internet</h1>

      <!-- create an unordered list -->
      <ul>
         <li>You can meet new people from countries around
            the world.</li>
         <li>
            You have access to new media as it becomes public:

            <!-- this starts a nested list, which uses a -->
```

Fig. 26.12 Nested and ordered lists in XHTML (part 1 of 3).

```
            <!-- modified bullet. The list ends when you -->
            <!-- close the <ul> tag.                     -->
            <ul>
               <li>New games</li>
               <li>
                  New applications

                  <!-- ordered nested list -->
                  <ol type = "I">
                     <li>For business</li>
                     <li>For pleasure</li>
                  </ol>
               </li>

               <li>Around the clock news</li>
               <li>Search engines</li>
               <li>Shopping</li>
               <li>
                  Programming

                  <!-- another nested ordered list -->
                  <ol type = "a">
                     <li>XML</li>
                     <li>Java</li>
                     <li>XHTML</li>
                     <li>Python</li>
                     <li>New languages</li>
                  </ol>

               </li>

            </ul> <!-- ends the nested list of line 27 -->
         </li>

         <li>Links</li>
         <li>Keeping in touch with old friends</li>
         <li>It is the technology of the future!</li>

      </ul>   <!-- ends the unordered list of line 18 -->

      <h1>My 3 Favorite <em>CEOs</em></h1>

      <!-- ol elements without a type attribute           -->
      <!-- have a numeric sequence type (i.e., 1, 2, ...) -->
      <ol>
         <li>Harvey Deitel</li>
         <li>Bill Gates</li>
         <li>Michael Dell</li>
      </ol>

   </body>
</html>
```

Fig. 26.12 Nested and ordered lists in XHTML (part 2 of 3).

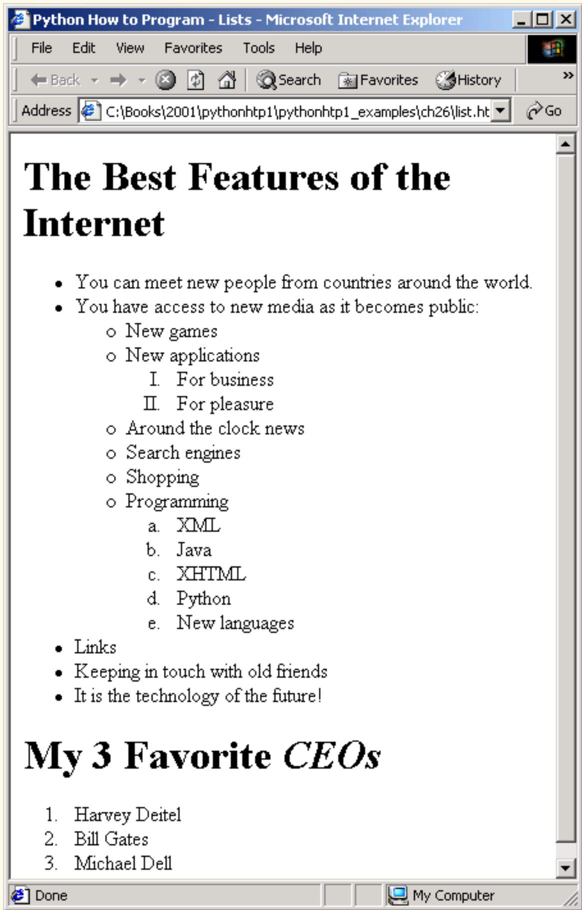


Fig. 26.12 Nested and ordered lists in XHTML (part 3 of 3).

The first ordered list begins on line 33. Attribute **type** specifies the *sequence type* (i.e., the set of numbers or letters used in the ordered list). In this case, setting **type** to **"I"** specifies upper-case Roman numerals. Line 46 begins the second ordered list and sets attribute **type** to **"a"**, specifying lowercase letters for the list items. The last ordered list (lines 64–68) does not use attribute **type**. By default, the list's items are enumerated from one to three.

A Web browser indents each nested list to indicate a hierarchal relationship. By default, the items in the outermost unordered list (line 18) are preceded by discs. List items nested inside the unordered list of line 18 are preceded by *circles*. Although not demonstrated in this example, subsequent nested list items are preceded by *squares*. Unordered list items may be explicitly set to discs, circles or squares by setting the **ul** element's **type** attribute to ***"disc"***, ***"circle"*** or ***"square"***, respectively.

Note: XHTML is based on HTML (HyperText Markup Language)—a legacy technology of the World Wide Web Consortium (W3C). In HTML, it was common to specify the document's content, structure and formatting. Formatting might specify where the

browser places an element in a Web page or the fonts and colors used to display an element. The so called *strict* form of XHTML allows only a document's content and structure to appear in a valid XHTML document, and not that document's formatting. Our first several examples used only the strict form of XHTML. In fact, the purpose of lines 2–3 in each of the examples before Fig. 26.12 was to indicate to the browser that each document conformed to the strict XHTML definition. This enables the browser to confirm that the document is valid. There are other XHTML document types as well. This particular example uses the XHTML *transitional* document type. This document type exists to enable XHTML document creators to use legacy HTML technologies in an XHTML document. In this example, the **`type`** attribute of the **`ol`** element (lines 33 and 46) is a legacy HTML technology. Changing lines 2–3 as shown in this example, enables us to demonstrate ordered lists with different numbering formats. Normally, such formatting is specified with style sheets (Chapter 28).

Testing and Debugging Tip 26.2

Most current browsers still attempt to render XHTML documents, even if they are invalid.

26.11 Internet and World Wide Web Resources

`www.w3.org/TR/xhtml1`
The *XHTML 1.0 Recommendation* contains XHTML 1.0 general information, compatibility issues, document type definition information, definitions, terminology and much more.

`www.xhtml.org`
XHTML.org provides XHTML development news and links to other XHTML resources, which include books and articles.

`www.w3schools.com/xhtml/default.asp`
The *XHTML School* provides XHTML quizzes and references. This page also contains links to XHTML syntax, validation and document type definitions.

`validator.w3.org`
This is the W3C XHTML validation service site.

`hotwired.lycos.com/webmonkey/00/50/index2a.html`
This site provides an article about XHTML. Key sections of the article overview XHTML and discuss tags, attributes and anchors.

`wdvl.com/Authoring/Languages/XML/XHTML`
The Web Developers Virtual Library provides an introduction to XHTML. This site also contains articles, examples and links to other technologies.

`www.w3.org/TR/1999/xhtml-modularization-19990406/DTD/doc`
The XHTML 1.0 DTD documentation site provides links to DTD documentation for the strict, transitional and frameset document type definitions.

SUMMARY

- XHTML (Extensible Hypertext Markup Language) is a markup language for creating Web pages.
- A key issue when using XHTML is the separation of the presentation of a document (i.e., the document's appearance when rendered by a browser) from the structure of the information in the document.

- In XHTML, text is marked up with elements, delimited by tags that are names contained in pairs of angle brackets. Some elements may contain additional markup called attributes, which provide additional information about the element.
- A machine that runs specialized piece of software called a Web server stores XHTML documents.
- XHTML documents that are syntactically correct are guaranteed to render properly. XHTML documents that contain syntax errors may not display properly.
- Validation services (e.g., **validator.w3.org**) ensure that an XHTML document is syntactically correct.
- Every XHTML document contains a start **<html>** tag and an end **</html>** tag.
- Comments in XHTML always begin with **<!--** and end with **-->**. The browser ignores all text inside a comment.
- Every XHTML document contains a **head** element, which generally contains information, such as a title, and a **body** element, which contains the page content. Information in the **head** element generally is not rendered in the display window but may be made available to the user through other means.
- The **title** element names a Web page. The title usually appears in the colored bar (called the title bar) at the top of the browser window and also appears as the text identifying a page when users add your page to their list of **Favorites** or **Bookmarks**.
- The body of an XHTML document is the area in which the document's content is placed. The content may include text and tags.
- All text placed between the **<p>** and **</p>** tags form one paragraph.
- XHTML provides six headers (**h1** through **h6**) for specifying the relative importance of information. Header element **h1** is considered the most significant header and is rendered in a larger font than the other five headers. Each successive header element (i.e., **h2**, **h3**, etc.) is rendered in a smaller font.
- Web browsers typically underline text hyperlinks and color them blue by default.
- The **<strong>** tag renders text in a bold font.
- Users can insert links with the **a** (anchor) element. The most important attribute for the **a** element is **href**, which specifies the resource (e.g., page, file, e-mail address, etc.) being linked.
- Anchors can link to an e-mail address using a **mailto:** URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Outlook Express) to initiate e-mail messages to the linked addresses.
- The **img** element's **src** attribute specifies an image's location. Optional attributes **width** and **height** specify the image width and height, respectively. Images are measured in pixels ("picture elements"), which represent dots of color on the screen. Every **img** element in a valid XHTML document must have an **alt** attribute, which contains text that is displayed if the client cannot render the image.
- The **alt** attribute makes Web pages more accessible to users with disabilities, especially those with vision impairments.
- Some XHTML elements are empty elements and contain only attributes and do not mark up text. Empty elements (e.g., **img**) must be terminated, either by using the forward slash character (**/**) or by explicitly writing an end tag.
- The **br** element causes most browsers to render a line break. Any markup or text following a **br** element is rendered on the next line.

- XHTML provides special characters or entity references (in the form **&*code*;**) for representing characters that cannot be marked up.
- Most browsers render a horizontal rule, indicated by the **<hr />** tag, as a horizontal line. The **hr** element also inserts a line break above and below the horizontal line.
- The unordered list element **ul** creates a list in which each item in the list begins with a bullet symbol (called a disc). Each entry in an unordered list is an **li** (list item) element. Most Web browsers render these elements with a line break and a bullet symbol at the beginning of the line.
- Lists may be nested to represent hierarchical data relationships.
- Attribute **type** specifies the sequence type (i.e., the set of numbers or letters used in the ordered list).

TERMINOLOGY

<!--…--> (XHTML comment)
a element (**<a>…**)
alt attribute
& (**&** special character)
anchor
angle brackets (**< >**)
attribute
body element
br (line break) element
comments in XHTML
© (**©** special character)
disc
element
e-mail anchor
empty tag
Extensible Hypertext Markup Language (XHTML)
head element
header
header elements (**h1** through **h6**)
height attribute
hexadecimal code
<hr /> tag (horizontal rule)
href attribute
.htm (XHTML file-name extension)
<html> tag
.html (XHTML file-name extension)
hyperlink
image hyperlink
img element
level of nesting
<li> (list item) tag
linked document
mailto: URL
markup language
nested list
ol (ordered list) element
p (paragraph) element

special character
src attribute (**img**)
<strong> tag
sub element
subscript
superscript
syntax
tag
text editor
text editor
title element
type attribute
unordered-list element (**ul**)
valid document
Web page
width attribute
World Wide Web (WWW)
XHTML (Extensible Hypertext Markup Language)
XHTML comment
XHTML markup
XHTML tag
XML declaration
xmlns attribute

SELF-REVIEW EXERCISES

26.1 State whether the following are *true* or *false*. If *false*, explain why.

a) Attribute **type**, when used with an **ol** element, specifies a sequence type.
b) An ordered list cannot be nested inside an unordered list.
c) XHTML is an acronym for XML HTML.
d) Element **br** represents a line break.
e) Hyperlinks are marked up with **<link>** tags.

26.2 Fill in the blanks in each of the following:

a) The __________ element inserts a horizontal rule.
b) A superscript is marked up using element __________ and a subscript is marked up using element __________.
c) The least important header element is __________ and the most important header element is __________.
d) Element __________ marks up an unordered list.
e) Element __________ marks up a paragraph.

ANSWERS TO SELF-REVIEW EXERCISES

26.1 a) True. b) False. An ordered list can be nested inside an unordered list. c) False. XHTML is an acronym for Extensible HyperText Markup Language. d) True. e) False. A hyperlink is marked up with **<a>** tags.

26.2 a) **hr**. b) **sup**, **sub**. c) **h6**, **h1**. d) **ul**. e) **p**.

EXERCISES

26.3 Use XHTML to create a document that contains the to mark up the following text:

Python How to Program
Welcome to the world of Python programming. We have provided extensive coverage on Python.

Use **h1** for the title (the first line of text), **p** for text (the second and third lines of text) and **sub** for each word that begins with a capital letter. Insert a horizontal rule between the **h1** element and the **p** element. Open your new document in a Web browser to view the marked up document.

26.4 Why is the following markup invalid?

```
<p>Here is some text...
<hr />
<p>And some more text...</p>
```

26.5 Why is the following markup invalid?

```
<p>Here is some text...<br>
And some more text...</p>
```

26.6 An image named **deitel.gif** is 200 pixels wide and 150 pixels high. Use the **width** and **height** attributes of the **<img>** tag to (a) increase the size of the image by 100%; (b) increase the size of the image by 50%; and (c) change the width-to-height ratio to 2:1, keeping the **width** attained in part (a). Write separate XHTML statements for parts (a), (b) and (c).

26.7 Create a link to each of the following: (a) **index.html**, located in the **files** directory; (b) **index.html**, located in the **text** subdirectory of the **files** directory; (c) **index.html**, located in the **other** directory in your *parent directory* [*Hint*: **..** signifies parent directory.]; (d) A link to the President of the United States' e-mail address (**president@whitehouse.gov**); and (e) An **FTP** link to the file named **README** in the **pub** directory of **ftp.cdrom.com** [*Hint*: Use **ftp://**.].

26.8 Create an XHTML document that marks up your resume.

26.9 Create an XHTML document containing three ordered lists: ice cream, soft serve and frozen yogurt. Each ordered list should contain a nested, unordered list of your favorite flavors. Provide a minimum of three flavors in each unordered list.

26.10 Create an XHTML document that uses an image as an e-mail link. Use attribute **alt** to provide a description of the image and link.

26.11 Create an XHTML document that contains an ordered list of your favorite Web sites. Your page should contain the header "My Favorite Web Sites."

26.12 Create an XHTML document that contains links to all the examples presented in this chapter. [*Hint*: Place all the chapter examples in one directory.]

26.13 Modify the XHTML document (**picture.html**) in Fig. 26.8 by removing all end tags. Validate this document using the W3C validation service. What happens? Next remove the **alt** attributes from the **<img>** tags and revalidate your document. What happens?

26.14 Identify each of the following as either an element or an attribute:

a) **html**.
b) **width**.
c) **href**.
d) **br**.
e) **h3**.
f) **a**.
g) **src**.

26.15 State which of the following statements are *true* and which are *false*. If *false*, explain why.
a) A valid XHTML document can contain uppercase letters in element names.
b) Tags need not be closed in a valid XHTML document.
c) XHTML documents can have the file extension **.htm**.
d) Valid XHTML documents can contain tags that overlap.
e) **&less;** is the special character for the less-than (**<**) character.
f) In a valid XHTML document, **<li>** can be nested inside either **<ol>** or **<ul>** tags.

26.16 Fill in the blanks for each of the following:
a) XHTML comments begin with **<!--** and end with ________.
b) In XHTML, attribute values must be enclosed in ________.
c) ________ is the special character for an ampersand.
d) Element ________ can be used to bold text.

(***DUMP FILE***)

SELF REVIEW EXERCISES

26.1 State whether the following are *true* or *false*. If *false*, explain why.
a) Attribute **type**, when used with an **ol** element, specifies a sequence type.
ANS: True.
b) An ordered list cannot be nested inside an unordered list.
ANS: False. An ordered list can be nested inside an unordered list.
c) XHTML is an acronym for XML HTML.
ANS: False. XHTML is an acronym for Extensible HyperText Markup Language.
d) Element **br** represents a line break.
ANS: True.
e) A hyperlink is marked up with **<link>** tags.
ANS: False. A hyperlink is marked up with **<a>** tags.

26.2 Fill in the blanks in each of the following:
a) The ________ element inserts a horizontal rule.
ANS: hr
b) A superscript is represented by element ________ and a subscript is represented by element ________.
ANS: sup, sub
c) The least important header element is ________ and the most important header element is ________.
ANS: h6, h1
d) Element ________ marks up an unordered list.
ANS: ul
e) Element ________ marks up a paragraph.
ANS: p

EXERCISES

26.3 Use XHTML to mark up the following text:

Python How to Program

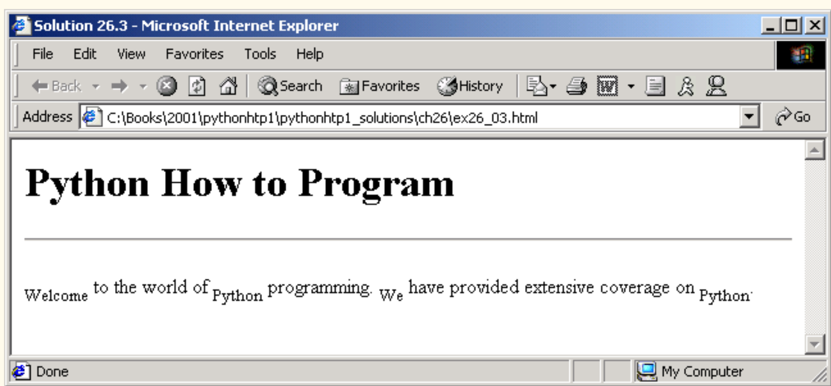
Welcome to the world of Python programming. We have provided extensive coverage on Python.
Use **h1** for the title, **p** for text and **sub** for each world that begins with a capital letter. Insert a horizontal rule between the **h1** element and the **p** element.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 26.3 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 26.3</title>
   </head>
   <body>
      <h1>Python How to Program</h1>
      <hr />
      <p><sub>Welcome</sub> to the world of <sub>Python</sub>
         programming. <sub>We</sub> have provided extensive
         coverage on <sub>Python</sub>.
      </p>
   </body>
</html>
```



26.4 Why is the following markup invalid?

```
<p>Here's some text...
<hr />
<p>And some more text...</p>
```

ANS: According to the XHTML specification, the `<p>` start tag must have a closing `</p>` tag.

26.5 Why is the following markup invalid?

```
<p>Here's some text...<br>
And some more text...</p>
```

ANS: According to the XHTML specification, the `<br>` tag must have a closing `</br>` tag or be written as an empty element `<br />`.

26.6 An image named **deitel.gif** that is 200 pixels wide and 150 pixels high. Use the **WIDTH** and **HEIGHT** attributes of the **IMG** tag to

a) increase image size by 100%;
ANS: <img src = "deitel.gif" width = "400" height = "300">

b) increase image size by 50%;
ANS: <img src = "deitel.gif" width = "300" height = "225">

c) change the width-to-height ratio to 2:1, keeping the width attained in a).
ANS: <img src = "deitel.gif" width = "400" height = "200">

26.7 Create a link to each of the following:

a) **index.html**, located in the **files** directory;
ANS: <a href = "files/index.html">

b) index.html, located in the **text** subdirectory of the **files** directory;
ANS: <a href = "files/text/index.html">

c) **index.html**, located in the **other** directory in your *parent directory* (*Hint*: **..** signifies parent directory.);
ANS: <a href = "../other/index.html">

d) A link to the President's email address (**president@whitehouse.gov**);
ANS: <a href = "mailto:president@whitehouse.gov">

e) An **FTP** link to the file named **README** in the **pub** directory of **ftp.cdrom.com** (*Hint*: remember to use **ftp://**).
ANS: <a href = "ftp://ftp.cdrom.com/pub/README">

26.8 Create an XHTML document that marks up your resume.
ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 26.8 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 26.8</title>
   </head>
   <body>

      <!-- heading of Resume -->

      <h1>John Doe<br /></h1>
      <h5>Boston College, PO Box 198, Chestnut Hill, MA 02167
         <br />
         Telephone: (617) 555-0000, E-mail<br />
         <a href= "mailto:deitel@bc.edu">deitel@bc.edu</a>
         <br />
      </h5>

      <hr /> <!-- inserts a horizontal rule -->

      <h2>Objective</h2>

      <!-- start of unordered lists -->
```

```
         <ul>
            <li>A position as an XHTML programmer</li>
         </ul>
         <h2>Education</h2>
         <ul>
            <li>Boston College, Chestnut Hill, MA<br />
               Computer Science Major, YOG 2003<br />
               Dean's List Fall 2001, Spring 2001</li>
         </ul>
         <h2>Skills</h2>
         <ul>
            <li>Computers</li>
            <li>Programming</li>
            <li>Typing, 55WPM</li>
               <ul>   <!-- start of nested list -->
                  <li>XHTML</li>
                  <li>Python</li>
                  <li>Cascading Style Sheets</li>
               </ul>   <!-- end of nested list -->

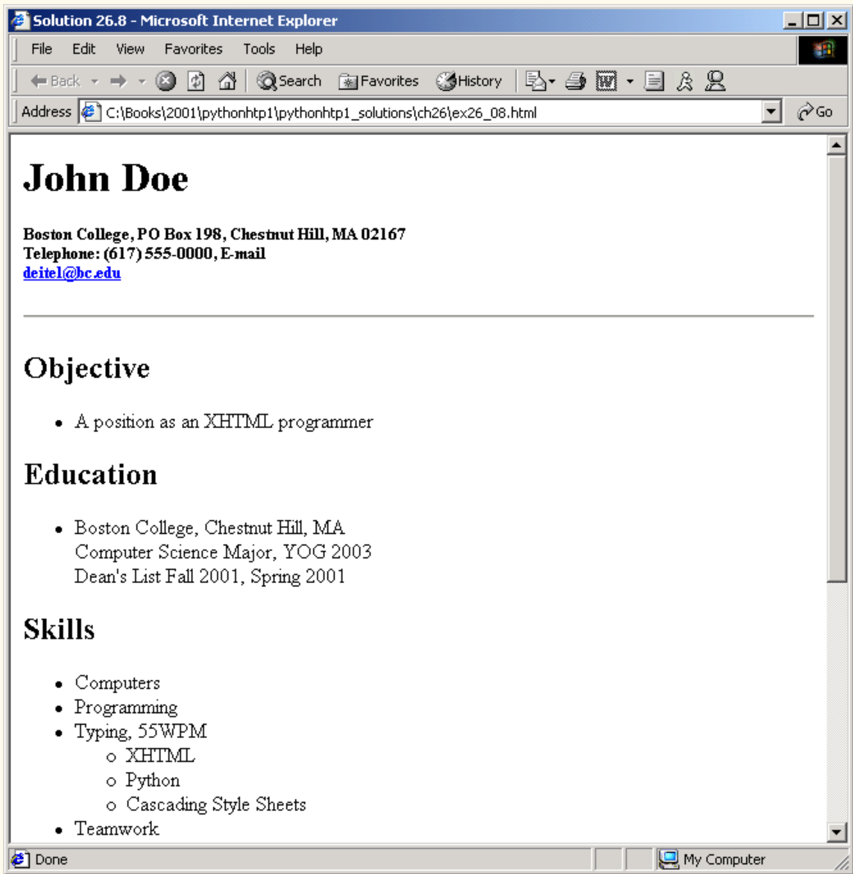
            </li>

            <li>Teamwork</li>
         </ul>
         <h2>Experience</h2>
         <ul>
            <li>Deitel & Associates,
               Sudbury, MA, Summer 2000</li>
            <li>Microsoft, Seattle, WA, Summer
               1999</li>
            <li>Computer Plus, Waltham, MA,
               Spring 1999</li>
         </ul>
         <h2>Interests and Activities</h2>
         <ul>
            <li>Soccer</li>
            <li>Guitar</li>
            <li>Music</li>
            <li>Student Government</li>
         </ul>

         <!-- end of unordered lists -->

         <hr />

      </body>
   </html>
```



26.9 Create an XHTML document containing three ordered lists: ice cream, soft serve and frozen yogurt. Each ordered list should contain a nested, unordered of your favorite flavors. Provide a minimum of three flavors in each unordered list.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 26.9 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 26.9</title>
   </head>
   <body>
      <h3>My Favorite Ice Cream, Soft Serve and Frozen Yogurt
```

```
            Flavors</h3>
         <!-- start of ordered list -->
         <ol>
            <li>Ice Cream

               <!-- start of nested unordered list -->
               <ul>
                  <li>Cherry Garcia</li>
                  <li>Cookie Dough</li>
                  <li>Bubble Gum</li>
                  <li>Coffee</li>
               </ul>

            </li>

            <li>Soft Serve

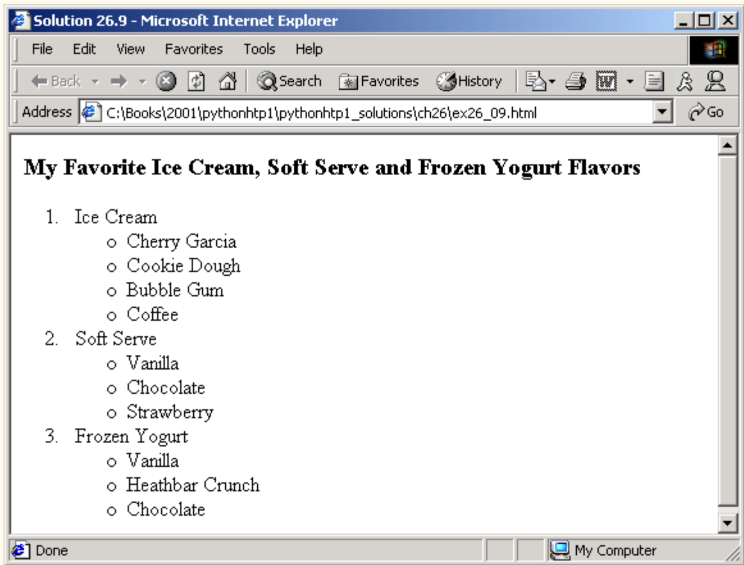
               <!-- another nested unordered list -->
               <ul>
                  <li>Vanilla</li>
                  <li>Chocolate</li>
                  <li>Strawberry</li>
               </ul>

            </li>

            <li>Frozen Yogurt

               <!-- another nested unordered list -->
               <ul>
                  <li>Vanilla</li>
                  <li>Heathbar Crunch</li>
                  <li>Chocolate</li>
               </ul>

            </li>
         </ol>
      </body>
   </html>
```



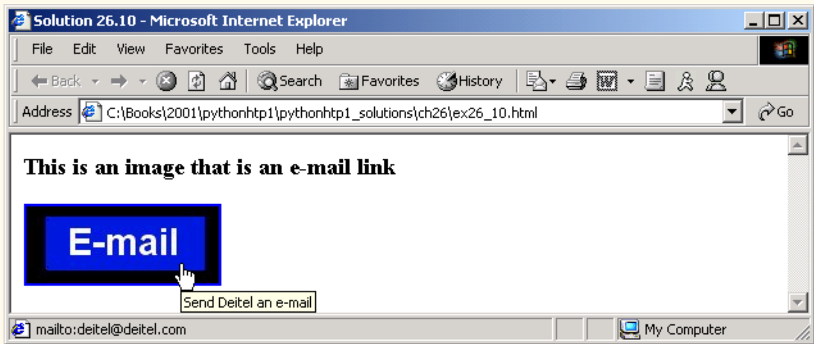
26.10 Create an XHTML document that uses an image as an e-mail link. Use attribute **`alt`** to provide a description of the image and link.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 26.10 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 26.10</title>
   </head>
   <body>
      <h3>This is an image that is an e-mail link</h3>
      <p>
         <!-- start of e-mail link with image -->
         <a href = "mailto:deitel@deitel.com">
            <img src = "email.jpg" alt = "Send Deitel
               an e-mail" /></a>
      </p>
   </body>
</html>
```



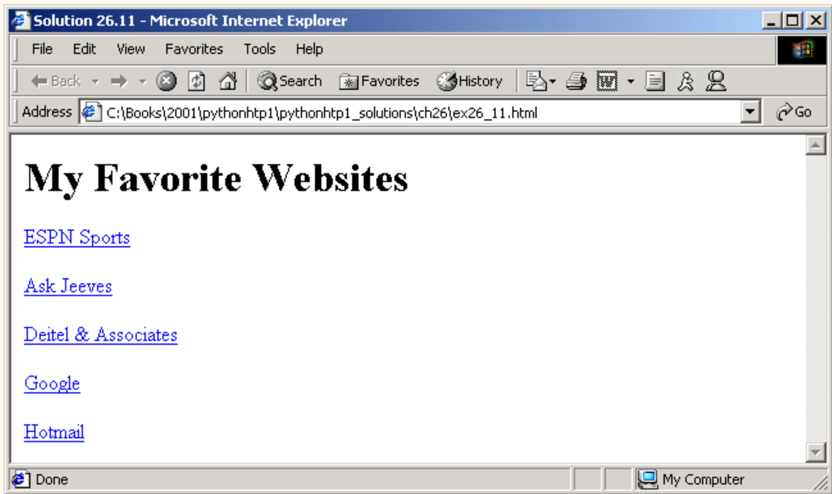
26.11 Create an XHTML document that contains an ordered list of your favorite Web sites. Your page should contain the header “My Favorite Web Sites.”

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 26.11 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 26.11</title>
   </head>
   <body>
      <h1>My Favorite Websites</h1>
      <p>
         <a href = "http://www.espn.com">ESPN Sports</a>
      </p>
      <p>
         <a href = "http://www.askjeeves.com">
            Ask Jeeves</a>
      </p>
      <p>
         <a href = "http://www.deitel.com">
            Deitel & Associates</a>
      </p>
      <p>
         <a href = "http://www.google.com">Google</a>
      </p>
      <p>
         <a href = "http://www.hotmail.com">Hotmail</a>
      </p>
   </body>
</html>
```



26.12 Create an XHTML document that contains links to all the examples presented in this chapter. [*Hint*: Place all the chapter examples in one directory.]

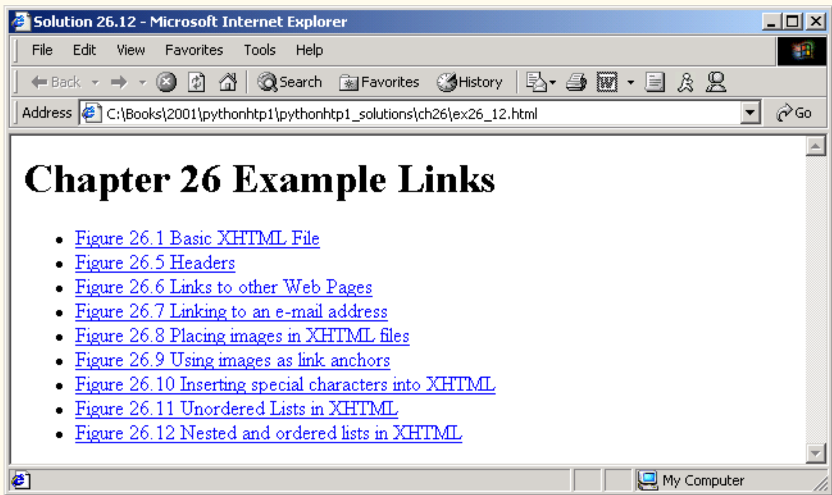
ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 26.12 Solution -->

<html>
   <head>
      <title>Solution 26.12</title>
   </head>
   <body>
      <h1>Chapter 26 Example Links</h1>
      <ul>
         <li><a href = "main.html">Figure 26.1 Basic XHTML File
            </a></li>
         <li><a href = "header.html">Figure 26.5 Headers
            </a></li>
         <li><a href = "links.html">Figure 26.6 Links to
            other Web Pages</a></li>
         <li><a href = "contact.html">Figure 26.7 Linking to an
            e-mail address</a></li>
         <li><a href = "picture.html">Figure 26.8 Placing images
            in XHTML files</a></li>
         <li><a href = "nav.html">Figure 26.9 Using images as
            link anchors</a></li>
         <li><a href = "contact2">Figure 26.10 Inserting special
            characters into XHTML</a></li>
         <li><a href = "links2.html">Figure 26.11 Unordered
```

```
                Lists in XHTML</a></li>
             <li><a href = "list">Figure 26.12 Nested and
                ordered lists in XHTML</a></li>
          </ul>
       </body>
    </html>
```



26.13 Modify the XHTML document (**picture.html**) in Fig. 26.8 by removing all end tags. Validate this document using the W3C validation service. What happens? Next remove the **alt** attributes from the **<img>** tags and re-validate your document. What happens?

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

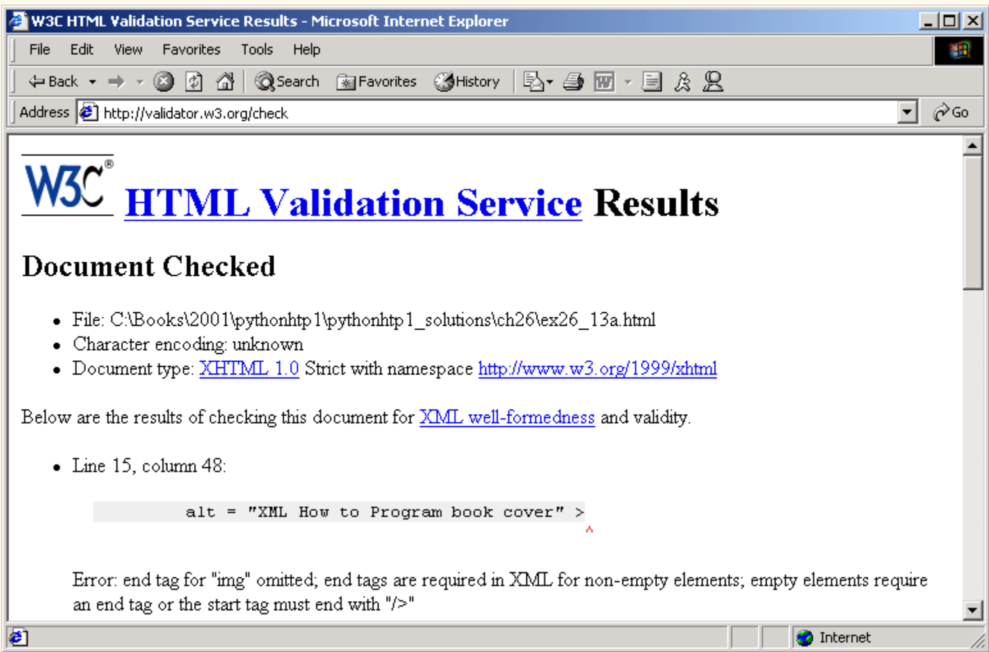
<!-- Exercise 26.13a Solution -->

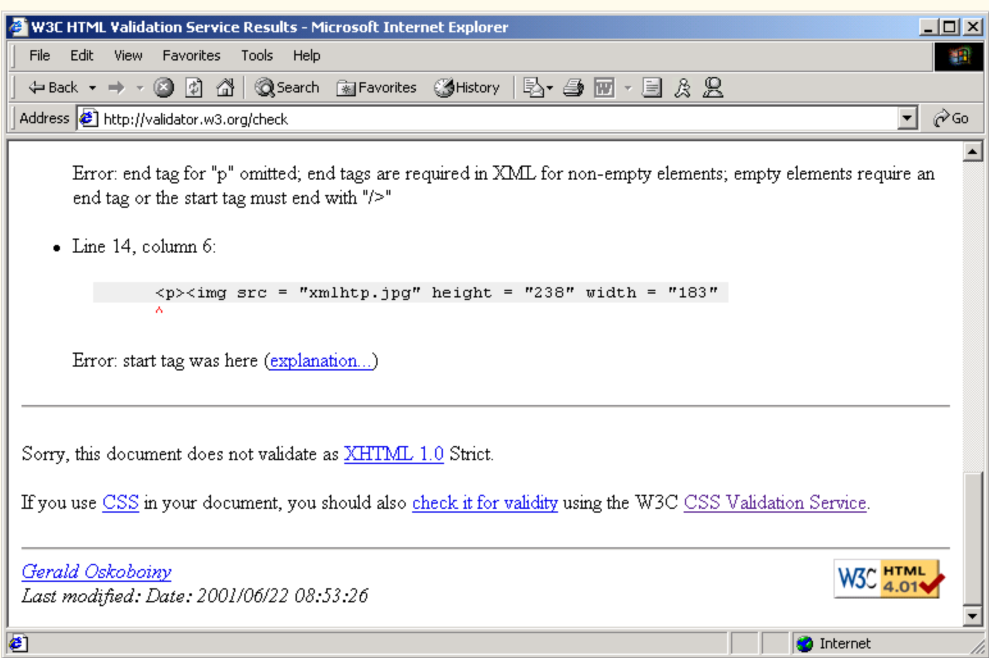
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Welcome</title>
   </head>

   <body>

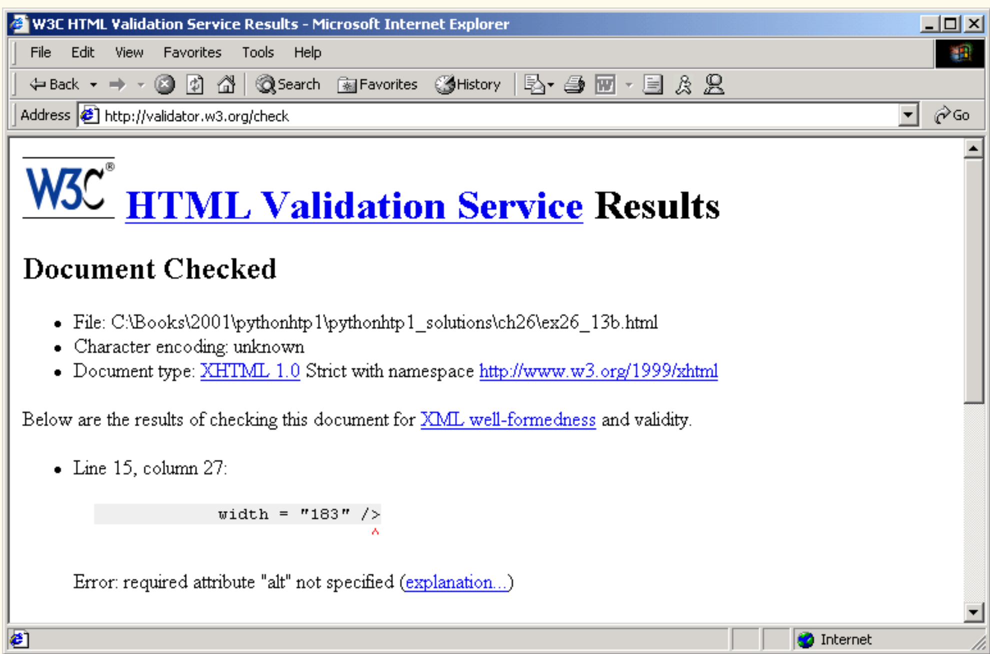
      <p><img src = "xmlhtp.jpg" height = "238" width = "183"
         alt = "XML How to Program book cover" >
         <img src = "jhtp.jpg" height = "238" width = "183"
```

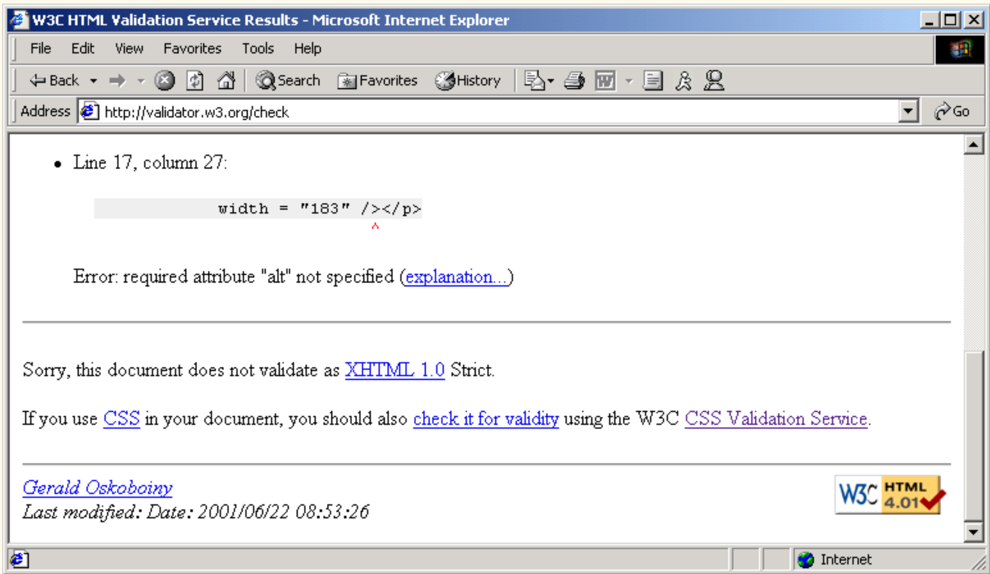
```
            alt = "Java How to Program book cover">
      </body>
   </html>
```





```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Exercise 26.13b Solution -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Python How to Program - Welcome</title>
10    </head>
11
12    <body>
13
14       <p><img src = "xmlhtp.jpg" height = "238"
15             width = "183" />
16          <img src = "jhtp.jpg" height = "238"
17             width = "183" /></p>
18    </body>
19 </html>
```





26.14 Identify each of the following as either an element or an attribute:

a) **`html`**
`ANS:` Element.
b) **`width`**
`ANS:` Attribute.
c) **`href`**
`ANS:` Attribute.
d) **`br`**
`ANS:` Element.
e) **`h3`**
`ANS:` Element.
f) **`a`**
`ANS:` Element.
g) **`src`**
`ANS:` Attribute.

26.15 State which of the following statements are *true* and which are *false*. If *false*, explain why.

a) A valid XHTML document can contain uppercase letters in element names.
`ANS:` False. All XHTML element names must be in lowercase.
b) Tags need not be closed in a valid XHTML document.
`ANS:` False. All XHTML tags are required to have corresponding closing tags.
c) XHTML can have the file extension **`.htm`**.
`ANS:` True.
d) Valid XHTML documents can contain tags that overlap.
`ANS:` False. XHTML prohibits overlapping tags.
e) **`&less;`** is the special character for the less-than (**<**) character.
`ANS:` False. **`<`** is the special character for less-than.
f) In a valid XHTML document, **`<li>`** can be nested inside either **`<ol>`** or **`<ul>`** tags.

ANS: True.

26.16 Fill in the blanks for each of the following:

a) XHTML comments begin with `<!--` and end with __________.

ANS: `-->`.

b) In XHTML, attribute values must be enclosed in __________.

ANS: quotes (single or double).

c) __________ is the special character for an ampersand.

ANS: `&`.

d) Element __________ can be used to bold text.

ANS: `strong`.

e) `<?xml version = "1.0"?>` is called the __________.

ANS: XML declaration.

[***Notes To Reviewers***]

- This chapter will be sent for second-round reviews.
- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send us e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **cheryl.yaeger@deitel.net**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
- Please be constructive. This book will be published soon. We all want to publish the best possible book.
- If you find something that is incorrect, please show us how to correct it.
- Please read all the back matter including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

S

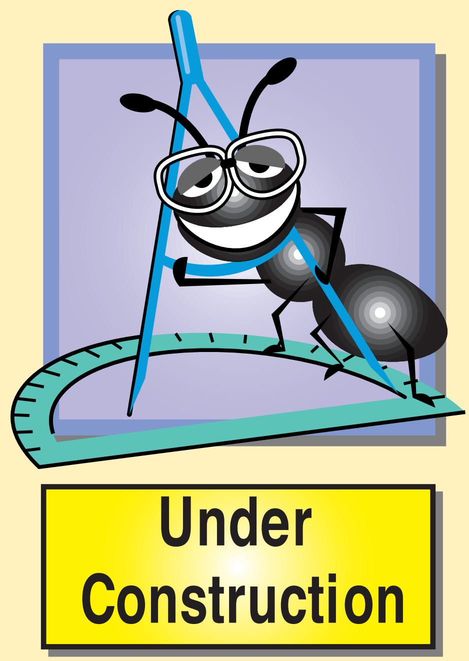
27

Introduction to XHTML: Part 2

Objectives

- To create tables with rows and columns of data.
- To control table formatting.
- To create and use forms.
- To create and use image maps to aid in Web-page navigation.
- To make Web pages accessible to search engines using **<meta>** tags.
- To se the **frameset** element to display multiple Web pages in a single browser window.

Yea, from the table of my memory
I'll wipe away all trivial fond records.
William Shakespeare



Outline

27.1 Introduction

In the previous chapter, we introduced XHTML. We built several complete Web pages featuring text, hyperlinks, images, horizontal rules and line breaks. In this chapter, we discuss more substantial XHTML features, including presentation of information in *tables* and *incorporating forms* for collecting information from a Web-page visitor. We also introduce *internal linking* and *image maps* for enhancing Web page navigation and *frames* for displaying multiple documents in the browser.

By the end of this chapter, you will be familiar with the most commonly used XHTML features and will be able to create more complex Web documents. In Chapter 28, we discuss how to make Web pages more visually appealing by manipulating fonts, colors and text.

27.2 Basic XHTML Tables

This section presents XHTML *tables*—a frequently used feature that organizes data into rows and columns. Our first example (Fig. 27.1) uses a table with six rows and two columns to display price information for fruit.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 27.1: table1.html  -->
<!-- Creating a basic table  -->

```

Fig. 27.1 XHTML table (part 1 of 3).

```
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>A simple XHTML table</title>
   </head>

   <body>

      <!-- the <table> tag opens a table -->
      <table border = "1" width = "40%"
         summary = "This table provides information about
            the price of fruit">

         <!-- the <caption> tag summarizes the table's    -->
         <!-- contents (this helps the visually impaired) -->
         <caption><strong>Price of Fruit</strong></caption>

         <!-- the <thead> is the first section of a table -->
         <!-- it formats the table header area            -->
         <thead>
            <tr>              <!-- <tr> inserts a table row -->
               <th>Fruit</th> <!-- insert a heading cell -->
               <th>Price</th>
            </tr>
         </thead>

         <!-- all table content is enclosed  -->
         <!-- within the <tbody>             -->
         <tbody>
            <tr>
               <td>Apple</td> <!-- insert a data cell -->
               <td>$0.25</td>
            </tr>

            <tr>
               <td>Orange</td>
               <td>$0.50</td>
            </tr>

            <tr>
               <td>Banana</td>
               <td>$1.00</td>
            </tr>

            <tr>
               <td>Pineapple</td>
               <td>$2.00</td>
            </tr>
         </tbody>

         <!-- the <tfoot> is the last section of a table -->
         <!-- it formats the table footer                -->
         <tfoot>
            <tr>
```

Fig. 27.1 XHTML table (part 2 of 3).

```
61                  <th>Total</th>
62                  <th>$3.75</th>
63               </tr>
64            </tfoot>
65
66         </table>
67
68      </body>
69   </html>
```

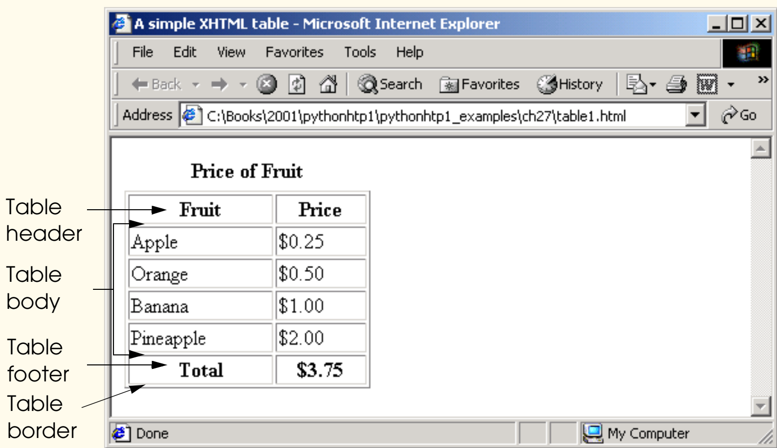


Fig. 27.1 XHTML table (part 3 of 3).

Tables are defined with the **table** element. Lines 16–18 specify the start tag for a table element that has several attributes. The ***border*** attribute specifies the table's border width in pixels. To create a table without a border, set **border** to **"0"**. This example assigns attribute **width "40%"** to set the table's width to 40 percent of the browser's width. A developer can also set attribute **width** to a specified number of pixels.

Testing and Debugging Tip 27.1

Try resizing the browser window to see how the width of the window affects the width of the table.

As its name implies, attribute ***summary*** (line 17) describes the table's contents. Speech devices use this attribute to make the table more accessible to users with visual impairments. The ***caption*** element (line 22) describes the table's content and helps text-based browsers interpret the table data. Text inside the **<caption>** tag is rendered above the table by most browsers. Attribute **summary** and element **caption** are two of many XHTML features that make Web pages more accessible to users with disabilities.

A table has three distinct sections—*head*, *body* and *foot*. The head section (or *header cell*) is defined with a ***thead*** element (lines 26–31), which contains header information such as column names. Each ***tr*** element (lines 27–30) defines an individual *table row*. The columns in the head section are defined with **th** elements. Most browsers center and display text formatted by **th** (table header column) elements in bold. Table header elements are nested inside table row elements.

The body section, or *table body*, contains the table's primary data. The table body (lines 35–55) is defined in a **tbody** element. *Data cells* contain individual pieces of data and are defined with **td** (*table data*) elements.

The foot section (lines 59–64) is defined with a ***tfoot*** (table foot) element and represents a footer. Common text placed in the footer includes calculation results and footnotes. Like other sections, the foot section can contain table rows and each row can contain columns.

27.3 Intermediate XHTML Tables and Formatting

In the previous section, we explored the structure of a basic table. In Fig. 27.2, we enhance our discussion of tables by introducing elements and attributes that allow the document author to build more complex tables.

The table begins on line 17. Element ***colgroup*** (lines 22–27) groups and formats columns. The ***col*** element (line 26) specifies two attributes in this example. The ***align*** attribute determines the alignment of text in the column. The ***span*** attribute determines how many columns the ***col*** element formats. In this case, we set **align**'s value to **"right"** and **span**'s value to **"1"** to right-align text in the first column (the column containing the picture of the camel in the sample screen capture).

Table cells are sized to fit the data they contain. Document authors can create large data cells by using attributes ***rowspan*** and ***colspan***. The values assigned to these attributes specify the number of rows or columns occupied by a cell. The **th** element in lines 36–39 uses the attribute **rowspan = "2"** to allow the cell containing the picture of the camel to use two vertically adjacent cells (thus the cell *spans* two rows). The **th** element in lines 42–45 uses the attribute **colspan = "4"** to widen the header cell (containing **Camelid comparison** and **Approximate as of 9/2002**) to span four cells.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 27.2: table2.html    -->
6  <!-- Intermediate table design -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Tables</title>
11    </head>
12
13    <body>
14
15       <h1>Table Example Page</h1>
16
17       <table border = "1">
18          <caption>Here is a more complex sample table.</caption>
19
20          <!-- <colgroup> and <col> tags are used to -->
```

Fig. 27.2 Complex XHTML table (part 1 of 3).

```
      <!-- format entire columns                 -->
      <colgroup>

         <!-- span attribute determines how many columns -->
         <!-- the <col> tag affects                      -->
         <col align = "right" span = "1" />
      </colgroup>

      <thead>

         <!-- rowspans and colspans merge the specified    -->
         <!-- number of cells vertically or horizontally   -->
         <tr>

            <!-- merge two rows -->
            <th rowspan = "2">
               <img src = "camel.gif" width = "205"
                  height = "167" alt = "Picture of a camel" />
            </th>

            <!-- merge four columns -->
            <th colspan = "4" valign = "top">
               <h1>Camelid comparison</h1><br />
               <p>Approximate as of 9/2002</p>
            </th>
         </tr>

         <tr valign = "bottom">
            <th># of Humps</th>
            <th>Indigenous region</th>
            <th>Spits?</th>
            <th>Produces Wool?</th>
         </tr>

      </thead>

      <tbody>

         <tr>
            <th>Camels (bactrian)</th>
            <td>2</td>
            <td>Africa/Asia</td>
            <td rowspan = "2">Llama</td>
            <td rowspan = "2">Llama</td>
         </tr>

         <tr>
            <th>Llamas</th>
            <td>1</td>
            <td>Andes Mountains</td>
         </tr>

      </tbody>
```

Fig. 27.2 Complex XHTML table (part 2 of 3).

```
74
75          </table>
76
77       </body>
78    </html>
```

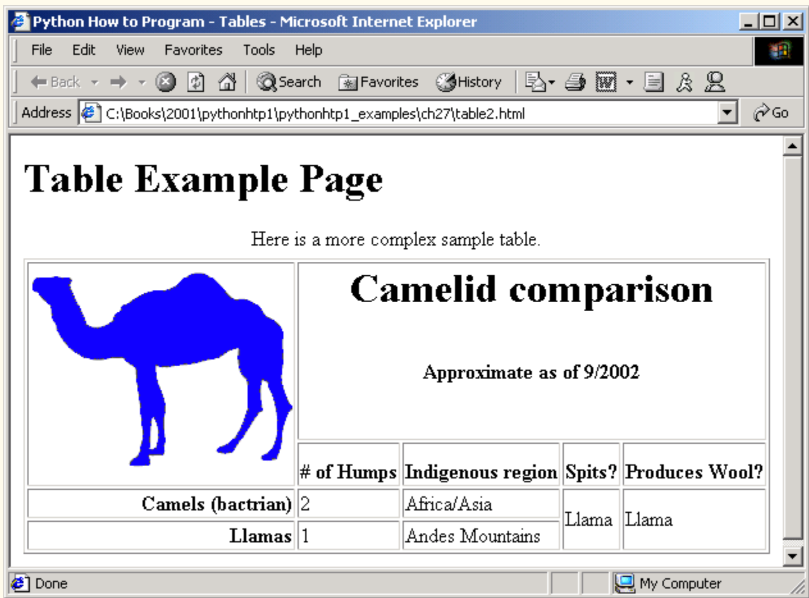


Fig. 27.2 Complex XHTML table (part 3 of 3).

Common Programming Error 27.1

When using **`colspan`** *and* **`rowspan`** *to adjust the size of table data cells, keep in mind that the modified cells will occupy more than one column or row. Other rows or columns of the table must compensate for the extra rows or columns spanned by individual cells. If you do not, the formatting of your table will be distorted and you may inadvertently create more columns and rows than you originally intended.*

Line 42 introduces attribute **`valign`**, which aligns data vertically and may be assigned one of four values—**`"top"`** aligns data with the top of the cell, **`"middle"`** vertically centers data (the default for all data and header cells), **`"bottom"`** aligns data with the bottom of the cell and **`"baseline"`** ignores the fonts used for the row data and sets the bottom of all text in the row on a common *baseline* (i.e., the horizontal line to which each character in a word is aligned).

27.4 Basic XHTML Forms

When browsing Web sites, users often need to provide information such as e-mail addresses, search keywords and zip codes. XHTML provides a mechanism, called a *form*, for collecting such user information.

Data that users enter on a Web page normally is sent to a Web server that provides access to a site's resources (e.g., XHTML documents, images, etc.). These resources are either located on the same machine as the Web server or on a machine that the Web server can access through the network. When a browser requests a Web page or file that is located on a server, the server processes the request and returns the requested resource. A request contains the name and path of the desired resource and the method of communication (called a *protocol*). XHTML documents use the HyperText Transfer Protocol (HTTP).

Figure 27.3 sends the form data to the Web server which passes the form data to a *CGI* (*Common Gateway Interface*) script (i.e., a program) written in Perl, C or some other language. The script processes the data received from the Web server and typically returns information to the Web server. The Web server then sends the information in the form of an XHTML document to the Web browser. [*Note*: This example demonstrates client-side functionality. If the form is submitted (by clicking **Submit Your Entries**) an error occurs. In later chapters such as Perl and Python, we present the server-side programming necessary to process information entered into a form.]

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 27.3: form.html   -->
<!-- Form Design Example 1 -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Forms</title>
   </head>

   <body>

      <h1>Feedback Form</h1>

      <p>Please fill out this form to help
         us improve our site.</p>

      <!-- this tag starts the form, gives the   -->
      <!-- method of sending information and the -->
      <!-- location of form scripts              -->
      <form method = "post" action = "/cgi-bin/formmail">

         <p>
            <!-- hidden inputs contain non-visual -->
            <!-- information                      -->
            <input type = "hidden" name = "recipient"

               value = "deitel@deitel.com" />
            <input type = "hidden" name = "subject"
               value = "Feedback Form" />
            <input type = "hidden" name = "redirect"
               value = "main.html" />
```

Fig. 27.3 Simple form with hidden fields and a text box (part 1 of 2).

```
34             </p>
35
36             <!-- <input type = "text"> inserts a text box -->
37             <p><label>Name:
38                   <input name = "name" type = "text" size = "25"
39                      maxlength = "30" />
40                </label></p>
41
42             <p>
43                <!-- input types "submit" and "reset" insert  -->
44                <!-- buttons for submitting and clearing the  -->
45                <!-- form's contents                          -->
46                <input type = "submit" value =
47                   "Submit Your Entries" />
48                <input type = "reset" value =
49                   "Clear Your Entries" />
50             </p>
51
52          </form>
53
54    </html>
```

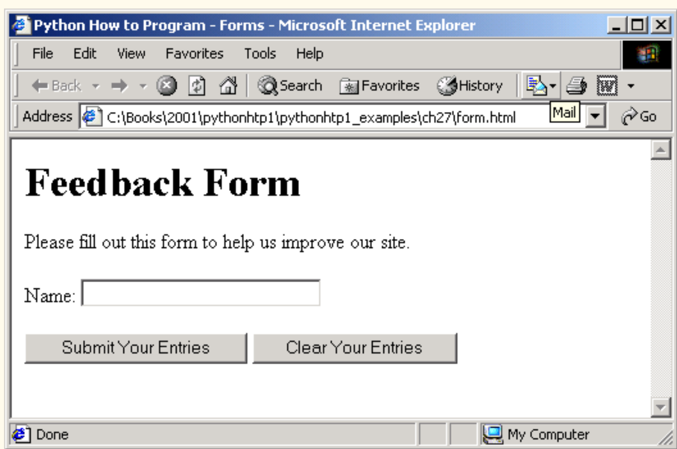


Fig. 27.3 Simple form with hidden fields and a text box (part 2 of 2).

Forms can contain visual and non-visual components. Visual components include clickable buttons and other graphical user interface components with which users interact. Non-visual components, called *hidden inputs*, store data that the document author specifies, such as e-mail addresses and XHTML document file names that act as links. The form begins on line 23 with the **`form`** element. Attribute **`method`** specifies how the form's data is sent to the Web server.

Using **`method = "post"`** appends form data to the browser request, which contains the protocol (i.e., HTTP) and the requested resource's URL. Scripts located on the Web server's computer (or on a computer accessible through the network) can access the form data sent as part of the request. For example, a script may take the form information and update an electronic mailing list. The other possible value, **`method = "get"`** appends the

form data directly to the end of the URL. For example, the URL **/cgi-bin/formmail** might have the form information **name = bob** appended to it.

The ***action*** attribute in the **<form>** tag specifies the URL of a script on the Web server; in this case, it specifies a script that e-mails form data to an address. Most *Internet Service Providers (ISPs)* have a script like this on their site; ask the Web site system administrator how to set up an XHTML document to use the script correctly.

Lines 28–33 define three ***input*** elements that specify data to provide to the script that processes the form (also called the *form handler*). These three **input** element have ***type*** attribute **"hidden"**, which allows the document author to send form data that is not entered by a user to a script.

The three hidden inputs are: an e-mail address to which the data will be sent, the e-mail's subject line and a URL where the browser will be redirected after submitting the form. Two other **input** attributes are ***name***, which identifies the **input** element, and ***value***, which provides the value that will be sent (or posted) to the Web server.

Good Programming Practice 27.1

*Place hidden **input** elements at the beginning of a form, immediately after the opening **<form>** tag. This placement allows document authors to locate hidden **input** elements quickly.*

We introduce another **type** of **input** in lines 38–39. The ***"text"*** **input** inserts a *text box* into the form. Users can type data in text boxes. The ***label*** element (lines 37–40) provides users with information about the **input** element's purpose.

Common Programming Error 27.2

*Forgetting to include a **label** element for each form element is a design error. Without these labels, users cannot determine the purpose of individual form elements.*

The **input** element's ***size*** attribute specifies the number of characters visible in the text box. Optional attribute ***maxlength*** limits the number of characters input into the text box. In this case, the user is not permitted to type more than **30** characters into the text box.

There are two types of **input** elements in lines 46–49. The ***"submit"*** **input** element is a button. When the user presses a **"submit"** button, the browser sends the data in the form to the Web server for processing. The ***value*** attribute sets the text displayed on the button (the default value is **Submit**). The ***"reset"*** **input** element allows a user to reset all **form** elements to their default values. The **value** attribute of the **"reset"** **input** element sets the text displayed on the button (the default value is **Reset**).

27.5 More Complex XHTML Forms

In the previous section, we introduced basic forms. In this section, we introduce elements and attributes for creating more complex forms. Figure 27.4 contains a form that solicits user feedback about a Web site.

The ***textarea*** element (lines 37–39) inserts a multiline text box, called a *text area*, into the form. The number of rows is specified with the ***rows*** *attribute* and the number of columns (i.e., characters) is specified with the ***cols*** *attribute*. In this example, the **textarea** is four rows high and 36 characters wide. To display default text in the text area,

place the text between the **<textarea>** and **</textarea>** tags. Default text can be specified in other **input** types, such as text boxes, by using the **value** attribute.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 27.4: form2.html -->
<!-- Form Design Example 2 -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - Forms</title>
   </head>

   <body>

      <h1>Feedback Form</h1>

      <p>Please fill out this form to help
         us improve our site.</p>

      <form method = "post" action = "/cgi-bin/formmail">

         <p>
            <input type = "hidden" name = "recipient"
               value = "deitel@deitel.com" />
            <input type = "hidden" name = "subject"
               value = "Feedback Form" />
            <input type = "hidden" name = "redirect"
               value = "main.html" />
         </p>

         <p><label>Name:
            <input name = "name" type = "text" size = "25" />
         </label></p>

         <!-- <textarea> creates a multiline textbox -->
         <p><label>Comments:<br />
            <textarea name = "comments" rows = "4" cols = "36">
               Enter your comments here.
            </textarea>
         </label></p>

         <!-- <input type = "password"> inserts a    -->
         <!-- textbox whose display is masked with  -->
         <!-- asterisk characters                   -->
         <p><label>E-mail Address:
            <input name = "email" type = "password"
               size = "25" />
         </label></p>

```

Fig. 27.4 Form with textareas, password boxes and checkboxes (part 1 of 3).

```
         <p>
            <strong>Things you liked:</strong><br />

            <label>Site design
            <input name = "thingsliked" type = "checkbox"
               value = "Design" /></label>

            <label>Links
            <input name = "thingsliked" type = "checkbox"
               value = "Links" /></label>

            <label>Ease of use
            <input name = "thingsliked" type = "checkbox"
               value = "Ease" /></label>

            <label>Images
            <input name = "thingsliked" type = "checkbox"
               value = "Images" /></label>

            <label>Source code
            <input name = "thingsliked" type = "checkbox"
               value = "Code" /></label>
         </p>

         <p>
            <input type = "submit" value =
               "Submit Your Entries" />
            <input type = "reset" value =
               "Clear Your Entries" />
         </p>

      </form>
</html>
```

Fig. 27.4 Form with textareas, password boxes and checkboxes (part 2 of 3).

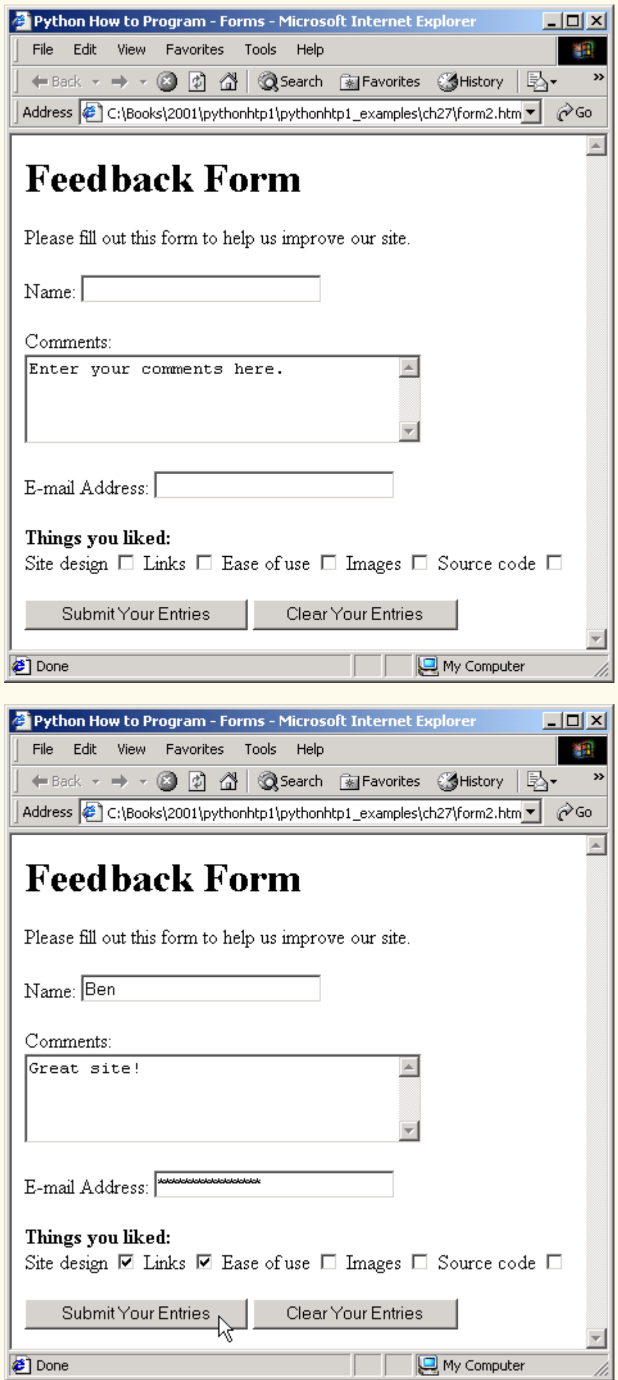


Fig. 27.4 Form with textareas, password boxes and checkboxes (part 3 of 3).

The **"password"** input in lines 46–47, inserts a password box with the specified **size**. A password box allows users to enter sensitive information, such as credit card numbers and passwords, by "masking" the information input with asterisks. The actual value input is sent to the Web server, not the asterisks that mask the input.

Lines 54–71 introduce the *checkbox* **form** element. Checkboxes enable users to select from a set of options. When a user selects a checkbox, a check mark appears in the check box. Otherwise, the checkbox remains empty. Each **"checkbox" input** creates a new checkbox. Checkboxes can be used individually or in groups. Checkboxes that belong to a group are assigned the same **name** (in this case, **"thingsliked"**).

Common Programming Error 27.3

*When your **form** has several checkboxes with the same **name**, you must make sure that they have different **value**s, or the scripts running on the Web server will not be able to distinguish between them.*

We continue our discussion of forms by presenting a third example that introduces several more form elements from which users can make selections (Fig. 27.5). In this example, we introduce two new **input** types. The first type is the *radio button* (lines 76–94) specified with type **"radio"**. Radio buttons are similar to checkboxes, except that only one radio button in a group of radio buttons may be selected at any time. All radio buttons in a group have the same **name** attributes and are distinguished by their different **value** attributes. The attribute-value pair **checked = "checked"** (line 77) indicates which radio button, if any, is selected initially. The **checked** attribute also applies to checkboxes.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 27.5: form3.html -->
6  <!-- Form Design Example 3 -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Forms</title>
11    </head>
12
13    <body>
14
15       <h1>Feedback Form</h1>
16
17       <p>Please fill out this form to help
18          us improve our site.</p>
19
20       <form method = "post" action = "/cgi-bin/formmail">
21
22          <p>
23             <input type = "hidden" name = "recipient"
24                value = "deitel@deitel.com" />
```

Fig. 27.5 Form including radio buttons and drop-down lists (part 1 of 5).

```
            <input type = "hidden" name = "subject"
               value = "Feedback Form" />
            <input type = "hidden" name = "redirect"
               value = "main.html" />
         </p>

         <p><label>Name:
               <input name = "name" type = "text" size = "25" />
            </label></p>

         <p><label>Comments:<br />
               <textarea name = "comments" rows = "4"
                  cols = "36"></textarea>
            </label></p>

         <p><label>E-mail Address:
               <input name = "email" type = "password"
                  size = "25" /></label></p>

         <p>
            <strong>Things you liked:</strong><br />

            <label>Site design
               <input name = "thingsliked" type = "checkbox"
                  value = "Design" /></label>

            <label>Links
               <input name = "thingsliked" type = "checkbox"
                  value = "Links" /></label>

            <label>Ease of use
               <input name = "thingsliked" type = "checkbox"
                  value = "Ease" /></label>

            <label>Images
               <input name = "thingsliked" type = "checkbox"
                  value = "Images" /></label>

            <label>Source code
               <input name = "thingsliked" type = "checkbox"
                  value = "Code" /></label>
         </p>

         <!-- <input type = "radio" /> creates a radio     -->
         <!-- button. The difference between radio buttons -->
         <!-- and checkboxes is that only one radio button -->
         <!-- in a group can be selected.                  -->
         <p>
            <strong>How did you get to our site?:</strong><br />

            <label>Search engine
               <input name = "howtosite" type = "radio"
                  value = "search engine" checked = "checked" />
```

Fig. 27.5 Form including radio buttons and drop-down lists (part 2 of 5).

```
            </label>

            <label>Links from another site
               <input name = "howtosite" type = "radio"
                  value = "link" /></label>

            <label>Deitel.com Web site
               <input name = "howtosite" type = "radio"
                  value = "deitel.com" /></label>

            <label>Reference in a book
               <input name = "howtosite" type = "radio"
                  value = "book" /></label>

            <label>Other
               <input name = "howtosite" type = "radio"
                  value = "other" /></label>

         </p>

         <p>
            <label>Rate our site:

                <!-- the <select> tag presents a drop-down -->
                <!-- list with choices indicated by the     -->
                <!-- <option> tags                          -->
               <select name = "rating">
                  <option selected = "selected">Amazing</option>
                  <option>10</option>
                  <option>9</option>
                  <option>8</option>
                  <option>7</option>
                  <option>6</option>
                  <option>5</option>
                  <option>4</option>
                  <option>3</option>
                  <option>2</option>
                  <option>1</option>
                  <option>Awful</option>
               </select>

            </label>
         </p>

         <p>
            <input type = "submit" value =
               "Submit Your Entries" />
            <input type = "reset" value = "Clear Your Entries" />
         </p>

      </form>

   </body>
```

Fig. 27.5 Form including radio buttons and drop-down lists (part 3 of 5).

```
</html>
```

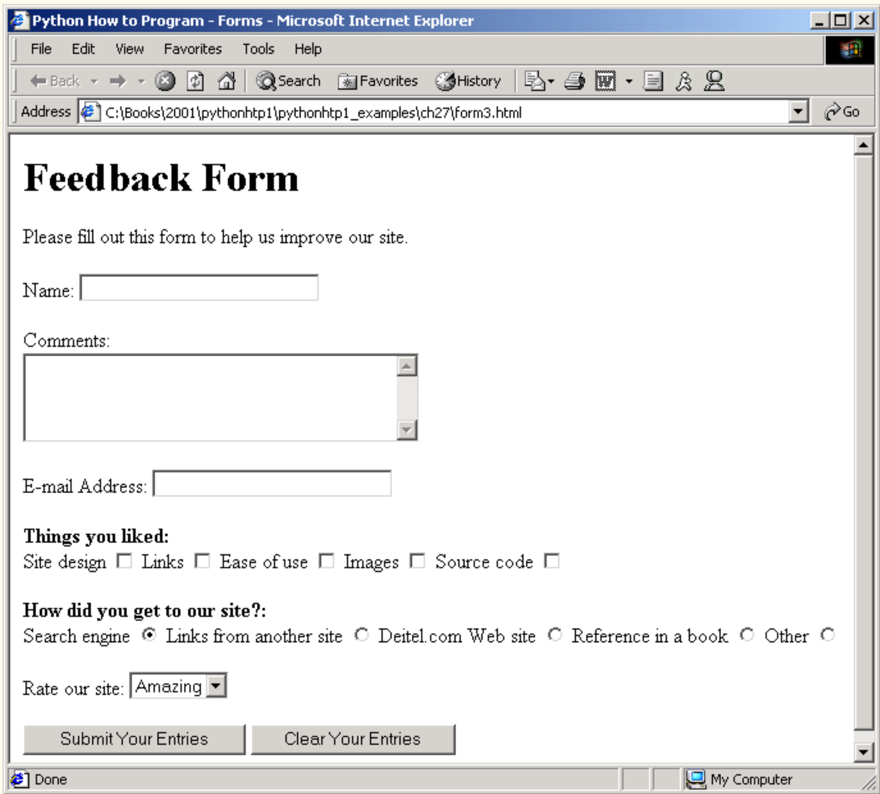


Fig. 27.5 Form including radio buttons and drop-down lists (part 4 of 5).

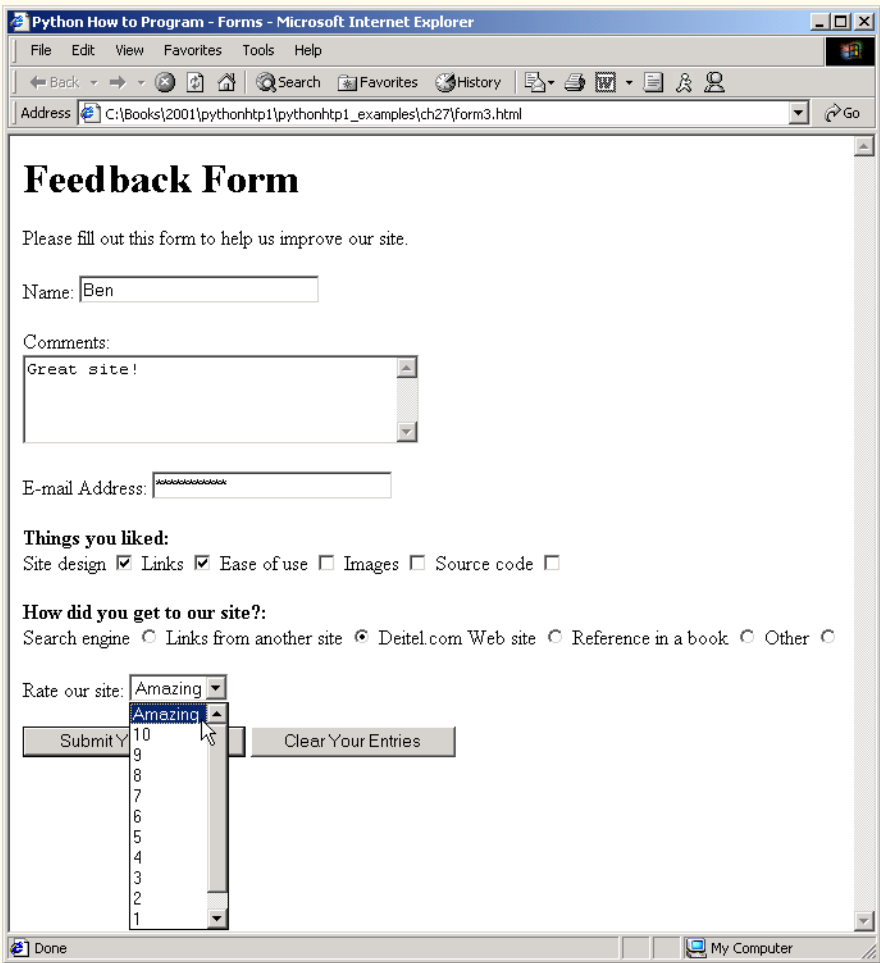


Fig. 27.5 Form including radio buttons and drop-down lists (part 5 of 5).

Common Programming Error 27.4

When using a group of radio buttons in a form, forgetting to set the ***name*** *attributes to the same name lets the user select all of the radio buttons at the same time, which is a logic error.*

The ***select*** element (lines 104–117) provides a drop-down list of items from which the user can select an item. The **name** attribute identifies the drop-down list. The ***option*** element (lines 105–116) adds items to the drop-down list. The **option** element's ***selected*** attribute specifies which item initially is displayed as the selected item in the **select** element.

27.6 Internal Linking

In Chapter 26, we discussed how to hyperlink one Web page to another. Figure 27.6 introduces *internal linking*—a mechanism that enables the user to jump between locations in the same document. Internal linking is useful for long documents that contain many sections. Clicking an internal link enables users to find a section without scrolling through the entire document.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 27.6: links.html -->
<!-- Internal Linking      -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Python How to Program - List</title>
   </head>

   <body>

      <!-- <a name = ".."></a> creates an internal hyperlink -->
      <p><a name = "features"></a></p>
      <h1>The Best Features of the Internet</h1>

      <!-- an internal link's address is "#linkname"         -->
      <p><a href = "#ceos">Go to <em>Favorite CEOs</em></a></p>

      <ul>
         <li>You can meet people from countries
            around the world.</li>

         <li>You have access to new media as it becomes public:
            <ul>
               <li>New games</li>
               <li>New applications
                  <ul>
                     <li>For Business</li>
                     <li>For Pleasure</li>
                  </ul>
               </li>

               <li>Around the clock news</li>
               <li>Search Engines</li>
               <li>Shopping</li>
               <li>Programming
                  <ul>
                     <li>XHTML</li>
                     <li>Java</li>
                     <li>Python</li>
```

Fig. 27.6 Using internal hyperlinks to make pages more navigable (part 1 of 3).

```
                     <li>Scripts</li>
                     <li>New languages</li>
                  </ul>
               </li>
            </ul>
         </li>

         <li>Links</li>
         <li>Keeping in touch with old friends</li>
         <li>It is the technology of the future!</li>
      </ul>

      <!-- named anchor -->
      <p><a name = "ceos"></a></p>
      <h1>My 3 Favorite <em>CEOs</em></h1>

      <p>

         <!-- internal hyperlink to features -->
         <a href = "#features">Go to <em>Favorite Features</em>
         </a></p>

      <ol>
         <li>Bill Gates</li>
         <li>Steve Jobs</li>
         <li>Michael Dell</li>
      </ol>

   </body>
</html>
```

Fig. 27.6 Using internal hyperlinks to make pages more navigable (part 2 of 3).

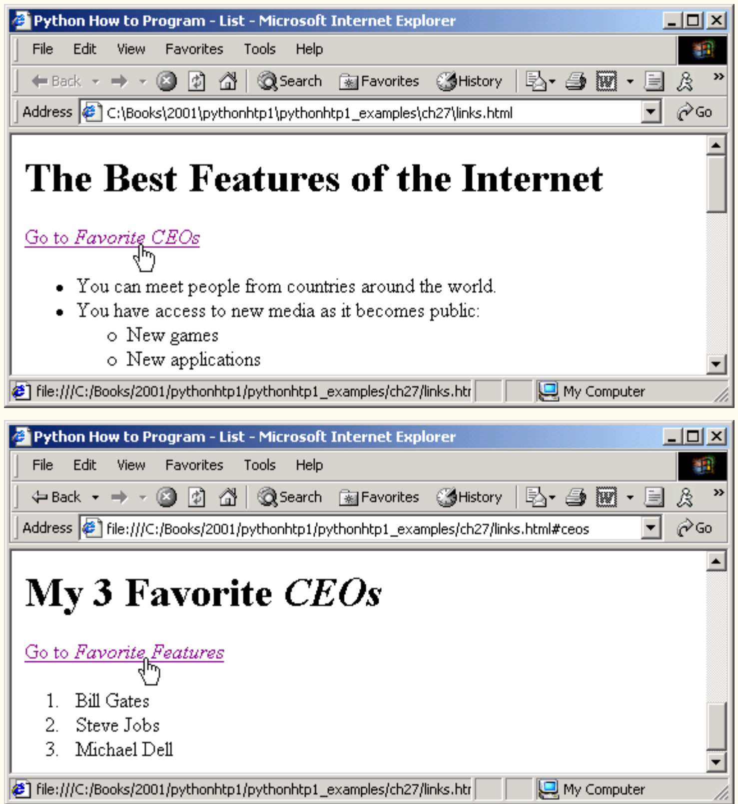


Fig. 27.6 Using internal hyperlinks to make pages more navigable (part 3 of 3).

Line 16 contains a *named anchor* (called **features**) for an internal hyperlink. To link to this type of anchor inside the same Web page, the href attribute of another anchor element includes the named anchor preceded with a pound sign (as in **#features**). Lines 63–64 contain a hyperlink with the anchor **features** as its target. Selecting this hyperlink in a Web browser scrolls the browser window to the **features** anchor at line 16.

Look-and-Feel Observation 27.1

Internal hyperlinks are useful in XHTML documents that contain large amounts of information. Internal links to various sections on the page makes it easier for users to navigate the page. They do not have to scroll to find a specific section.

Although not demonstrated in this example, a hyperlink can specify an internal link in another document by specifying the document name followed by a pound sign and the named anchor, as in:

```
href = "page.html#name"
```

For example, to link to a named anchor called **booklist** in **books.html**, **href** is assigned **"books.html#booklist"**.

27.7 Creating and Using Image Maps

In Chapter 26, we demonstrated how images can be used as hyperlinks to link to other resources on the Internet. In this section, we introduce another technique for image linking called *image maps*, which designate certain areas of an image (called *hotspots*) as links. Figure 27.7 introduces image maps and hotspots.

```
<?xml version = "1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 27.7: picture.html       -->
<!-- Creating and Using Image Maps -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>
         Python How to Program - Image Map
      </title>
   </head>

   <body>

      <p>

      <!-- the <map> tag defines an image map -->
      <map id = "picture">

         <!-- shape = "rect" indicates a rectangular    -->
         <!-- area, with coordinates for the upper-left -->
         <!-- and lower-right corners                  -->
         <area href = "form.html" shape = "rect"
            coords = "2,123,54,143"
            alt = "Go to the feedback form" />
         <area href = "contact.html" shape = "rect"
            coords = "126,122,198,143"
            alt = "Go to the contact page" />
         <area href = "main.html" shape = "rect"
            coords = "3,7,61,25" alt = "Go to the homepage" />
         <area href = "links.html" shape = "rect"
            coords = "168,5,197,25"
            alt = "Go to the links page" />

         <!-- value "poly" creates a hotspot in the shape -->
         <!-- of a polygon, defined by coords             -->
         <area shape = "poly" alt = "E-mail the Deitels"
            coords = "162,25,154,39,158,54,169,51,183,39,161,26"
            href = "mailto:deitel@deitel.com" />

         <!-- shape = "circle" indicates a circular -->
         <!-- area with the given center and radius -->
         <area href = "mailto:deitel@deitel.com"
```

Fig. 27.7 Image with links anchored to an image map (part 1 of 2).

```
46               shape = "circle" coords = "100,36,33"
47               alt = "E-mail the Deitels" />
48         </map>
49
50         <!-- <img src =... usemap = "#id"> indicates that the -->
51         <!-- specified image map is used with this image     -->
52         <img src = "deitel.gif" width = "200" height = "144"
53            alt = "Deitel logo" usemap = "#picture" />
54         </p>
55      </body>
56   </html>
```

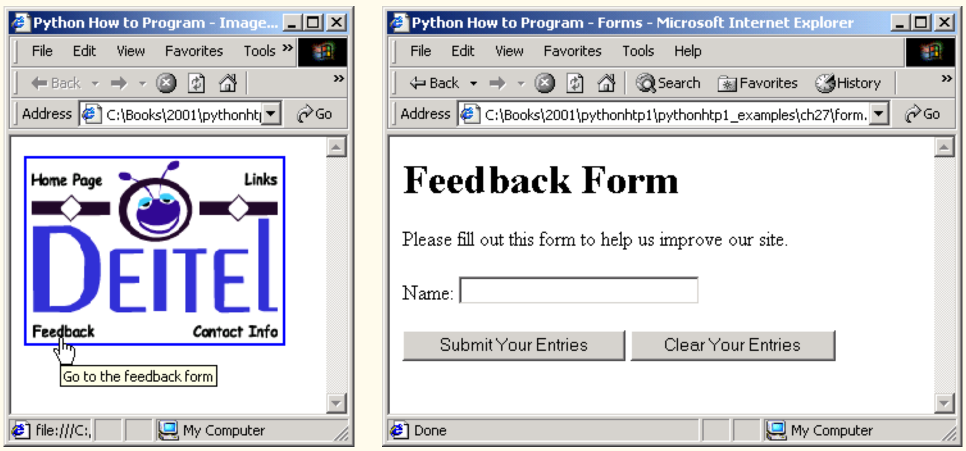


Fig. 27.7 Image with links anchored to an image map (part 2 of 2).

Lines 20–48 define image maps using a ***map*** element. Attribute ***id*** (line 20) identifies the image map. If **id** is omitted, the map cannot be referenced by an image. We discuss how to reference an image map momentarily. Hotspots are defined with ***area*** elements (as shown on lines 25–27). Attribute **href** (line 25) specifies the link's target (i.e., the resource to which to link). Attributes ***shape*** (line 25) and ***coords*** (line 26) specify the hotspot's shape and coordinates, respectively. Attribute **alt** (line 27) provides alternate text for the link.

Common Programming Error 27.5

Not specifying an ***id*** *attribute for a* ***map*** *element prevents an* ***img*** *element from using the* ***map****'s* ***area*** *elements to define hotspots.*

The markup on lines 25–27 creates a *rectangular hotspot* (**shape = "rect"**) for the *coordinates* specified in the **coords** attribute. A coordinate pair consists of two numbers representing the location of a point on the *x*-axis and the *y*-axis, respectively. The *x*-axis extends horizontally and the *y*-axis extends vertically from the upper-left corner of the image. Every point on an image has a unique *x*-*y*-coordinate. For rectangular hotspots, the required coordinates are those of the upper-left and lower-right corners of the rectangle. In this case, the upper-left corner of the rectangle is located at 2 on the *x*-axis and 123 on the

y-axis, annotated as *(2, 123)*. The lower-right corner of the rectangle is at *(54, 143)*. Coordinates are measured in pixels.

Common Programming Error 27.6

Overlapping coordinates of an image map cause the browser to render the first hotspot it encounters for the area.

The map **area** (lines 39–41) assigns the **shape** attribute ***"poly"*** to create a hotspot in the shape of a polygon using the coordinates in attribute **coords**. These coordinates represent each *vertex*, or corner, of the polygon. The browser connects these points with lines to form the hotspot's area.

The map **area** (lines 45–47) assigns the **shape** attribute ***"circle"*** to create a *circular hotspot*. In this case, the **coords** attribute specifies the circle's center coordinates and the circle's radius, in pixels.

To use an image map with an **img** element, the **img** element's ***usemap*** attribute is assigned the **id** of a **map**. Lines 52–53 reference the image map named **"picture"**. The image map resides within the same document, so we use internal linking.

27.8 meta Elements

People use search engines to find useful Web sites. Search engines usually catalog sites by following links from page to page and saving identification and classification information for each page. One way that search engines catalog pages is by reading the content in each page's ***meta*** elements, which specify information about a document.

Two important attributes of the **meta** element are ***name***, which identifies the type of **meta** element and ***content***, which provides the information search engines use to catalog pages. Figure 27.8 introduces the **meta** element.

Lines 14–16 demonstrate a ***"keywords"*** **meta** element. The ***content*** attribute of such a **meta** element provides search engines with a list of words that describe a page. These words are compared with words in search requests. Thus, including **meta** elements and their **content** information exposes Web sites to a wider audience.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 27.8: main.html -->
6  <!-- <meta> tag           -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Welcome</title>
11
12       <!-- <meta> tags provide search engines with -->
13       <!-- information used to catalog a site      -->
14       <meta name = "keywords" content = "Web page, design,
15          XHTML, tutorial, personal, help, index, form,
```

Fig. 27.8 Using **meta** to provide keywords and a description (part 1 of 2).

```
16            contact, feedback, list, links, frame, deitel" />
17
18         <meta name = "description" content = "This Web site will
19            help you learn the basics of XHTML and Web page design
20            through the use of interactive examples and
21            instruction." />
22
23      </head>
24
25      <body>
26
27         <h1>Welcome to Our Web Site!</h1>
28
29         <p>We have designed this site to teach about the wonders
30         of <strong><em>XHTML</em></strong>. <em>XHTML</em> is
31         better equipped than <em>HTML</em> to represent complex
32         data on the Internet. <em>XHTML</em> takes advantage of
33         XML's strict syntax to ensure well-formedness. Soon you
34         will know about many of the great new features of
35         <em>XHTML.</em></p>
36
37        <p>Have Fun With the Site!</p>
38
39      </body>
40   </html>
```

Fig. 27.8 Using **meta** to provide keywords and a description (part 2 of 2).

Lines 18–21 demonstrate a ***"description"*** **meta** element. The **content** attribute of such a **meta** element provides a three- to four-line description of a site, written in sentence form. Search engines also use this description to catalog your site and sometimes display this information as part of the search results.

Software Engineering Observation 27.1

meta *elements are not visible to users and must be placed inside the* ***head*** *section of your XHTML document. If* ***meta*** *elements are not placed in this section, they will not be read by search engines.*

27.9 frameset Element

All of the Web pages we have presented in this book have the ability to link to other pages, but can display only one page at a time. Figure 27.9 uses *frames*, which allow the browser to display more than one XHTML document simultaneously, to display the documents in Fig. 27.8 and Fig. 27.10.

Most of our prior examples conformed to the strict XHTML document type. This particular example uses the *frameset* document type—a special XHTML document type spe-

cifically for framesets. This new document type is specified in lines 2–3 and is required for documents that define framesets.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
4
5  <!-- Fig. 27.9: index.html -->
6  <!-- XHTML Frames I        -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Main</title>
11       <meta name = "keywords" content = "Webpage, design,
12          XHTML, tutorial, personal, help, index, form,
13          contact, feedback, list, links, frame, deitel" />
14
15       <meta name = "description" content = "This Web site will
16          help you learn the basics of XHTML and Web page design
17          through the use of interactive examples
18          and instruction." />
19
20    </head>
21
22    <!-- the <frameset> tag sets the frame dimensions     -->
23    <frameset cols = "110,*">
24
25       <!-- frame elements specify which pages -->
26       <!-- are loaded into a given frame      -->
27       <frame name = "leftframe" src = "nav.html" />
28       <frame name = "main" src = "main.html" />
29
30       <noframes>
31          <p>This page uses frames, but your browser does not
32          support them.</p>
33
34          <p>Please, <a href = "nav.html">follow this link to
35          browse our site without frames</a>.</p>
36       </noframes>
37
38    </frameset>
39 </html>
```

Fig. 27.9 Web document containing two frames—navigation and content (part 1 of 2).

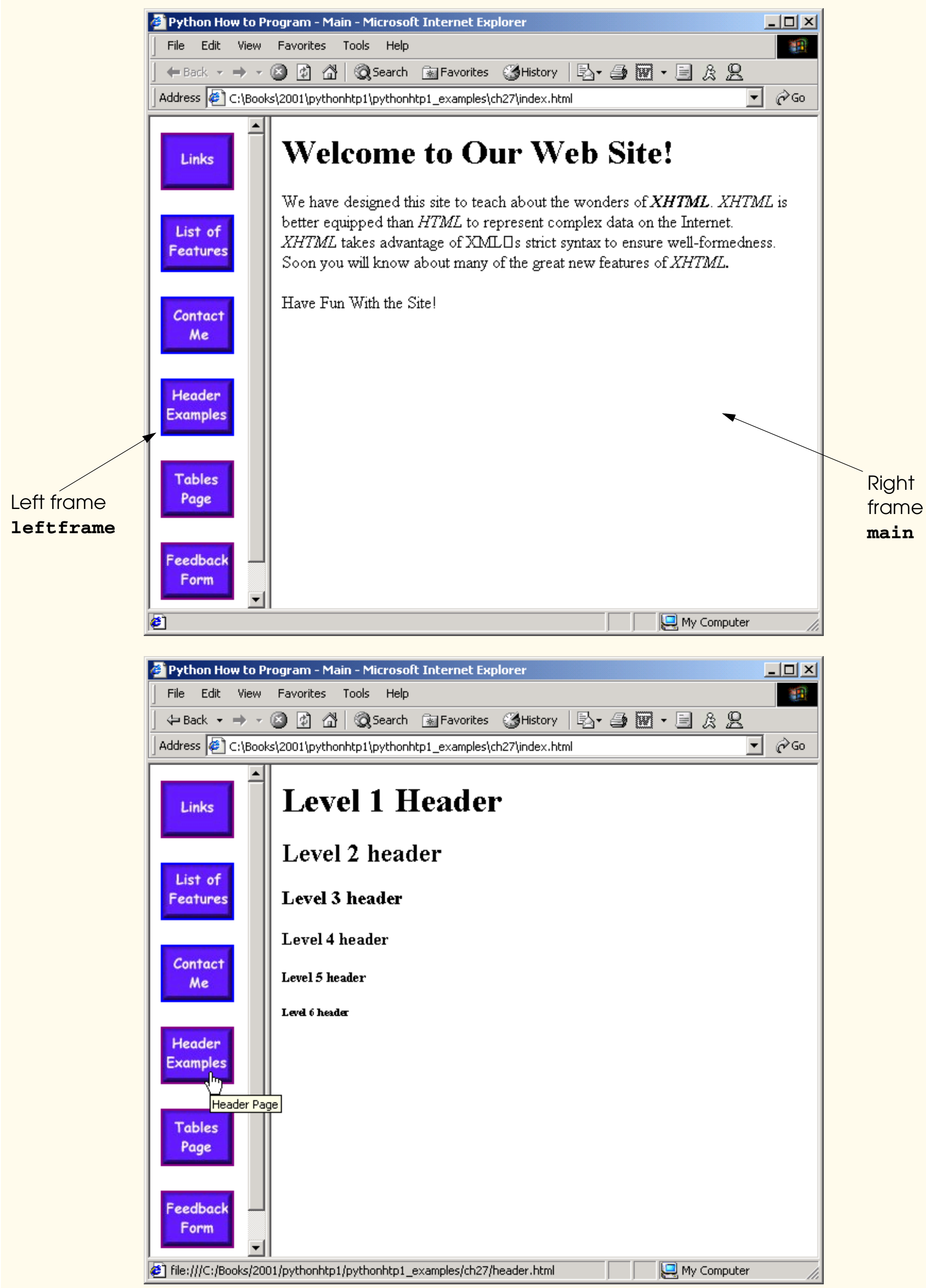


Fig. 27.9 Web document containing two frames—navigation and content (part 2 of 2).

A document that defines a frameset normally consists of an **html** element that contains a **head** element and a ***frameset*** element. The ***<frameset>*** tag (line 23) informs the browser that a page contains frames. Attribute ***cols*** specifies the frameset's column layout. The value of **cols** gives the width of each frame, either in pixels or as a percentage of the browser width. In this case, the attribute **cols = "110,*"** informs the browser that there are two vertical frames. The first frame extends **110** pixels from the left edge of the browser window and the second frame fills the remainder of the browser width (as indicated by the asterisk). Similarly, **frameset** attribute ***rows*** specifies the number of rows and the size of each row in a frameset.

The documents that will be loaded into the **frameset** are specified with ***frame*** elements (lines 27–28 in this example). Attribute **src** specifies the URL of the page to display in the frame. Each frame has **name** and **src** attributes. The first frame (which covers **110** pixels on the left side of the **frameset**) is named **leftframe** and displays the page **nav.html** (Fig. 27.10). The second frame is named main and displays the page **main.html**.

Attribute **name** identifies a frame, enabling hyperlinks in a **frameset** to specify the ***target*** **frame** in which a linked document should display when the user clicks the link. For example

```
<a href = "links.html" target = "main">
```

loads **links.html** in the frame whose **name** is **"main"**.

Not all browsers support frames. XHTML provides the ***noframes*** element (lines 30–36) to enable XHTML document designers to specify alternate content for browsers that do not support frames.

Portability Tip 27.1

Some browsers do not support frames. Use the ***noframes*** *element inside a* ***frameset*** *to direct users to a nonframed version of your site.*

Fig. 27.10 is the Web page displayed in the left frame of Fig. 27.9. This XHTML document provides the navigation buttons that, when clicked, determine which document is displayed in the right frame.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 27.10: nav.html          -->
<!-- Using images as link anchors -->

<html xmlns = "http://www.w3.org/1999/xhtml">

   <head>
      <title>Python How to Program - Navigation Bar
      </title>
   </head>

```

Fig. 27.10 XHTML document displayed in the left frame of Fig. 27.9 (part 1 of 2).

```
15       <body>
16 
17          <p>
18             <a href = "links.html" target = "main">
19                <img src = "buttons/links.jpg" width = "65"
20                   height = "50" alt = "Links Page" />
21             </a><br />
22 
23             <a href = "list.html" target = "main">
24                <img src = "buttons/list.jpg" width = "65"
25                   height = "50" alt = "List Example Page" />
26             </a><br />
27 
28             <a href = "contact.html" target = "main">
29                <img src = "buttons/contact.jpg" width = "65"
30                   height = "50" alt = "Contact Page" />
31             </a><br />
32 
33             <a href = "header.html" target = "main">
34                <img src = "buttons/header.jpg" width = "65"
35                   height = "50" alt = "Header Page" />
36             </a><br />
37 
38             <a href = "table1.html" target = "main">
39                <img src = "buttons/table.jpg" width = "65"
40                   height = "50" alt = "Table Page" />
41             </a><br />
42 
43             <a href = "form.html" target = "main">
44                <img src = "buttons/form.jpg" width = "65"
45                   height = "50" alt = "Feedback Form" />
46             </a><br />
47          </p>
48 
49       </body>
50    </html>
```

Fig. 27.10 XHTML document displayed in the left frame of Fig. 27.9 (part 2 of 2).

Line 27 (Fig. 27.9) displays the XHTML page in Fig. 27.10. Anchor attribute **target** (line 18 in Fig. 27.10) specifies that the linked documents are loaded in frame **main** (line 28 in Fig. 27.9). A **target** can be set to a number of preset values: **"_blank"** loads the page into a new browser window, **"_self"** loads the page into the frame in which the anchor element appears and **"_top"** loads the page into the full browser window (i.e., removes the **frameset**).

27.10 Nested framesets

You can use the **frameset** element to create more complex layouts in a Web page by nesting **frameset**s, as in Fig. 27.11. The nested **frameset** in this example displays the XHTML documents in Fig. 27.7, Fig. 27.8 and Fig. 27.10.

The outer frameset element (lines 23–41) defines two columns. The left frame extends over the first 110 pixels from the left edge of the browser and the right frame occupies the rest of the window's width. The **frame** element on line 24 specifies that the document **nav.html** (Fig. 27.10) displays in the left column.

Lines 28–31 define a nested **frameset** element for the second column of the outer frameset. This **frameset** defines two rows. The first row extends 175 pixels from the top of the browser window, as indicated by **rows = "175,*"**. The second row occupies the remainder of the browser window's height. The **frame** element at line 29 specifies that the first row of the nested **frameset** displays **picture.html** (Fig. 27.7). The **frame** element at line 30 specifies that the second row of the nested **frameset** displays **main.html** (Fig. 27.9).

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
4
5  <!-- Fig. 27.11: index2.html -->
6  <!-- XHTML Frames II         -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Python How to Program - Main</title>
11
12       <meta name = "keywords" content = "Webpage, design,
13          XHTML, tutorial, personal, help, index, form,
14          contact, feedback, list, links, frame, deitel" />
15
16       <meta name = "description" content = "This Web site will
17          help you learn the basics of XHTML and Web page design
18          through the use of interactive examples
19          and instruction." />
20
21    </head>
22
23    <frameset cols = "110,*">
24       <frame name = "leftframe" src = "nav.html" />
25
26       <!-- nested framesets are used to change the -->
27       <!-- formatting and layout of the frameset  -->
28       <frameset rows = "175,*">
29          <frame name = "picture" src = "picture.html" />
30          <frame name = "main" src = "main.html" />
31       </frameset>
32
33       <noframes>
34          <p>This page uses frames, but your browser does not
35          support them.</p>
36
37          <p>Please, <a href = "nav.html">follow this link to
38          browse our site without frames</a>.</p>
```

Fig. 27.11 Framed Web site with a nested frameset (part 1 of 2).

```
         </noframes>

      </frameset>
   </html>
```



Fig. 27.11 Framed Web site with a nested frameset (part 2 of 2).

Testing and Debugging Tip 27.2

When using nested **`frameset`** *elements, indent every level of* **`<frame>`** *tag. This practice makes the page clearer and easier to debug.*

In this chapter, we presented XHTML for marking up information in tables, creating forms for gathering user input, linking to sections within the same document, using **`<meta>`** tags and creating frames. In Chapter 28, we build upon the XHTML introduced in this chapter by discussing how to make Web pages more visually appealing with Cascading Style Sheets.

27.11 Internet and World Wide Web Resources

`courses.e-survey.net.au/xhtml/index.html`
The *Web Page Design - XHTML* site provides descriptions and examples for various XHTML features, such as links, tables, frames, forms, etc. Users can e-mail questions or comments to the Web Page Design support staff.

www.vbxml.com/xhtml/articles/xhtml_tables
The *VBXML.com* Web site contains a tutorial on creating XHTML tables.

www.webreference.com/xml/reference/xhtml.html
This Web page contains a list of the frequently used XHTML tags, such as header tags, table tags, frame tags and form tags. It also provides a description of each tag.

SUMMARY

- XHTML tables mark up tabular data and are one of the most frequently used features in XHTML.
- The **table** element defines an XHTML table. Attribute **border** specifies the table's border width, in pixels. Tables without borders set this attribute to **"0"**.
- Element **summary** summarizes the table's contents and is used by speech devices to make the table more accessible to users with visual impairments.
- Element **caption** describe's the table's content. The text inside the **<caption>** tag is rendered above the table in most browsers.
- A table can be split into three distinct sections: head (**thead**), body (**tbody**) and foot (**tfoot**). The head section contains information such as table titles and column headers. The table body contains the primary table data. The table foot contains information such as footnotes.
- Element **tr**, or table row, defines individual table rows. Element **th** defines a header cell. Text in **th** elements usually is centered and displayed in bold by most browsers. This element can be present in any section of the table.
- Data within a row are defined with **td**, or table data, elements.
- Element **colgroup** groups and formats columns. Each **col** element can format any number of columns (specified with the **span** attribute).
- The document author has the ability to merge data cells with the **rowspan** and **colspan** attributes. The values assigned to these attributes specify the number of rows or columns occupied by the cell. These attributes can be placed inside any data-cell tag.
- XHTML provides forms for collecting information from users. Forms contain visual components such as buttons that users click. Forms may also contain non-visual components, called hidden inputs, which store data, such as e-mail addresses and XHTML document file names used for linking.
- A form begins with the **form** element. Attribute **method** specifies how the form's data is sent to the Web server.
- The **"text"** input inserts a text box into the form. Text boxes allow the user to input data.
- The **input** element's **size** attribute specifies the number of characters visible in the **input** element. Optional attribute **maxlength** limits the number of characters input into a text box.
- The **"submit"** input submits the data entered in the form to the Web server for processing. Most Web browsers create a button that submits the form data when clicked. The **"reset"** input allows a user to reset all **form** elements to their default values.
- The **textarea** element inserts a multiline text box, called a text area, into a form. The number of rows in the text area is specified with the **rows** attribute and the number of columns (i.e., characters) is specified with the **cols** attribute.
- The **"password"** input inserts a password box into a form. A password box allows users to enter sensitive information, such as credit card numbers and passwords, by "masking" the information input with another character. Asterisks are the masking character used for password boxes. The actual value input is sent to the Web server, not the asterisks that mask the input.

- The checkbox input allows the user to make a selection. When the checkbox is selected, a check mark appears in the check box. Otherwise, the checkbox is empty. Checkboxes can be used individually or in groups. Checkboxes that are part of the same group have the same **name**.
- A radio button is similar in function to a checkbox, except that only one radio button in a group can be selected at any time. All radio buttons in a group have the same **name** attribute value and have different attribute **value**s.
- The **select** input provides a drop-down list of items. The **name** attribute identifies the drop-down list. The **option** element adds items to the drop-down list. The **selected** attribute, like the **checked** attribute for radio buttons and checkboxes, specifies which list item is displayed initially.
- Image maps designate certain sections of an image as links. These links are more properly called hotspots.
- Image maps are defined with **map** elements. Attribute **id** identifies the image map. Hotspots are defined with the **area** element. Attribute **href** specifies the link's target. Attributes **shape** and **coords** specify the hotspot's shape and coordinates, respectively, and **alt** provides alternate text.
- One way that search engines catalog pages is by reading the **meta** elements's contents. Two important attributes of the **meta** element are **name**, which identifies the type of **meta** element and **content**, which provides information a search engine uses to catalog a page.
- Frames allow the browser to display more than one XHTML document simultaneously. The **frameset** element informs the browser that the page contains frames. Not all browsers support frames. XHTML provides the **noframes** element to specify alternate content for browsers that do not support frames.
- You can use the **frameset** element to create more complex layouts in a Web page by nesting **frameset**s.

TERMINOLOGY

action attribute
area element
border attribute
browser request
<caption> tag
checkbox
checked attribute
col element
colgroup element
cols attribute
colspan attribute
coords element
form
form element
frame element
frameset element
header cell
hidden input element
hotspot
href attribute
image map

img element
input element
internal hyperlink
internal linking
map element
maxlength attribute
meta element
method attribute
name attribute
navigational frame
nested **frameset** element
nested tag
noframes element
password box
"radio" (attribute value)
rows attribute (**textarea**)
rowspan attribute (**tr**)
selected attribute
size attribute (**input**)
table element
target = "_blank"
target = "_self"
target = "_top"
tbody element
td element
textarea
textarea element
tfoot (table foot) element
<thead>...</thead>
tr (table row) element
type attribute
usemap attribute
valign attribute (**th**)
value attribute
Web server
XHTML form
x-*y*-coordinate

SELF-REVIEW EXERCISES

27.1 State whether the following statements are *true* or *false*. If *false*, explain why.

a) The width of all data cells in a table must be the same.
b) Framesets can be nested.
c) You are limited to a maximum of 100 internal links per page.
d) All browsers can render **frameset**s.

27.2 Fill in the blanks in each of the following statements:

a) Assigning attribute **type** __________ in an **input** element inserts a button that, when clicked, clears the contents of the form.
b) The layout of a **frameset** is set by including the __________ attribute or the __________ attribute inside the **<frameset>** tag.

c) The __________ element marks up a table row.
d) __________ are used as masking characters in a password box.
e) The common shapes used in image maps are __________, __________ and __________.

27.3 Write XHTML markup to accomplish each of the following:
a) Insert a framed Web page, with the first frame extending 300 pixels across the page from the left side.
b) Insert a table with a border of 8.
c) Indicate alternate content to a **`frameset`**.
d) Insert an image map in a page using **`deitel.gif`** as an image and **`map`** with **`name = "hello"`** as the image map, and set the **`alt`** text to "**`hello`**".

ANSWERS TO SELF-REVIEW EXERCISES

27.1 a) False. You can specify the width of any column, either in pixels or as a percentage of the table width. b) True. c) False. You can have an unlimited number of internal links. d) False. Some browsers are unable to render a **`frameset`** and must therefore rely on the information that you include inside the **`<noframes>`**…**`</noframes>`** tags.

27.2 a) **`"reset"`**. b) **`cols`**, **`rows`**. c) **`tr`**. d) asterisks. e) **`poly`** (polygons), **`circle`**s, **`rect`** (rectangles).

27.3 a) `<frameset cols = "300,*">…</frameset>`
b) `<table border = "8">…</table>`
c) `<noframes>…</noframes>`
d) `<img src = "deitel.gif" alt = "hello" usemap = "#hello" />`

EXERCISES

27.4 Categorize each of the following as an element or an attribute:
a) **`width`**.
b) **`td`**.
c) **`th`**.
d) **`frame`**.
e) **`name`**.
f) **`select`**.
g) **`type`**.

27.5 What will the **`frameset`** produced by the following code look like? Assume that the pages referenced are blank with white backgrounds and that the dimensions of the screen are 800 by 600. Sketch the layout, approximating the dimensions.

```
<frameset rows = "20%,*">
   <frame src = "hello.html" name = "hello" />
      <frameset cols = "150,*">
         <frame src = "nav.html" name = "nav" />
         <frame src = "deitel.html" name = "deitel" />
      </frameset>
</frameset>
```

27.6 Write the XHTML markup to create a frame with a table of contents on the left side of the window, and have each entry in the table of contents use internal linking to scroll down the document frame to the appropriate subsection.

27.7 Create XHTML markup that produces the table shown in Fig. 27.12. Use **<em>** and **<strong>** tags as necessary. The image (**camel.gif**) is included in the Chapter 27 examples directory on the CD-ROM that accompanies this book.

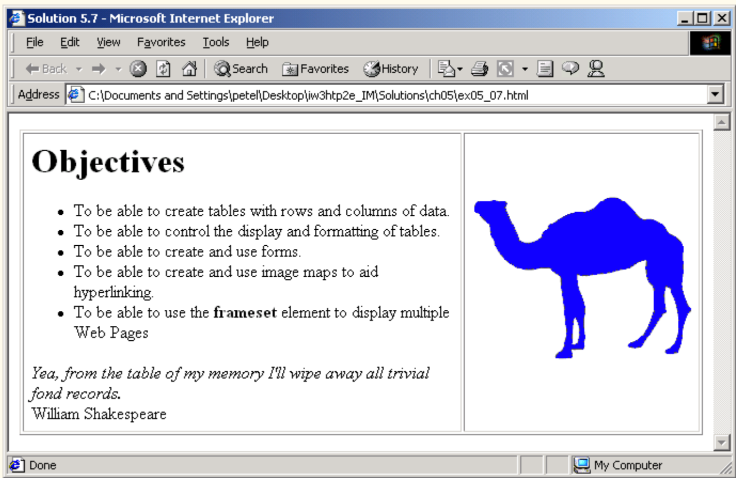


Fig. 27.12 XHTML table for Exercise 27.7.

27.8 Write an XHTML document that produces the table shown in Fig. 27.13.

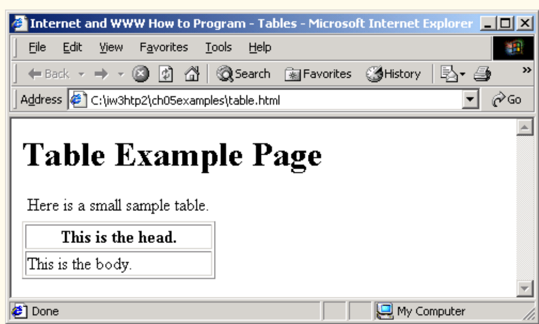


Fig. 27.13 XHTML table for Exercise 27.8.

27.9 A local university has asked you to create an XHTML document that allows potential students to provide feedback about their campus visits. Your XHTML document should contain a form with text boxes for names, addresses and e-mails. Provide check boxes that allow prospective students to indicate what they liked most about the campus. These check boxes should include: students, location, campus, atmosphere, dorm rooms and sports. Also, provide radio buttons that ask the prospective student how they became interested in the university. Options should include: friends, television, Internet and other. In addition, provide a text area for additional comments, a submit button and a reset button.

27.10 Create an XHTML document titled "How to Get Good Grades." Use **<meta>** tags to include a series of keywords that describe your document.

27.11 Create an XHTML document that displays a tic-tac-toe table with player X winning. Use **<h2>** to mark up both Xs and Os. Center the letters in each cell horizontally. Title the game using an **<h1>** tag. This title should span all three columns. Set the table border to one.

*(***DUMP FILE***)*

SELF-REVIEW EXERCISES

27.1 State whether the following are *true* or *false*. If *false*, explain why.

a) The width of all data cells in a table must be the same.

ANS: False. You can specify the width of any column, either in pixels or as a percentage of the table width.

b) Framesets can be nested.

ANS: True.

c) You are limited to a maximum of 100 internal links per page.

ANS: False. You can have an unlimited number of internal links.

d) All browsers can render **frameset**s.

ANS: False. Some browsers are unable to render a **frameset** and must therefore rely on the information that you include inside the **<noframes>**…**</noframes>** tags.

27.2 Fill in the blanks in each of the following statements:

a) The ________ attribute in an **input** element inserts a button that, when clicked, clears the contents of the form.

ANS: **type = "reset"**.

b) The spacing of a **frameset** is set by including the ________ attribute or the ________ attribute inside the **<frameset>** tag.

ANS: **cols**, **rows**.

c) The ________ element marks up a table row.

ANS: **tr**.

d) ________ are typically used as masking characters in a password box.

ANS: Asterisks.

e) The common shapes used in image maps are ________, ________ and ________.

ANS: **poly**, **circle**, **rect**.

27.3 Write XHTML markup to accomplish the following:

a) Insert a framed Web page, with the first frame extending 300 pixels across the page from the left side.

ANS: **<frameset cols = "300,*">**…**</frameset>**

b) Insert a table with a border of 8.

ANS: **<table border = "8">**…**</table>**

c) Indicate alternate content to a **frameset**.

ANS: **<noframes>**…**</noframes>**

d) Insert an image map in a page using **deitel.gif** as an image and **map** with **name = "hello"** as the image map, and set the **alt** text to "**hello**".

ANS: **.**

EXERCISES

27.4 Categorize each of the following as an element or an attribute:

a) width

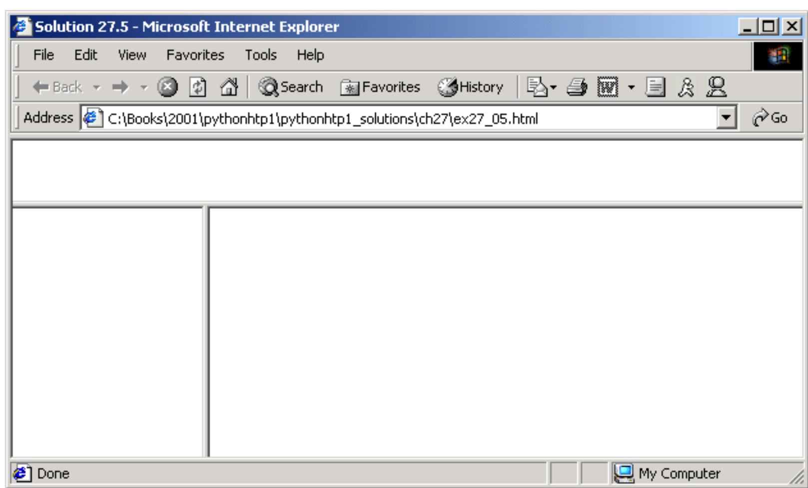
ANS: Attribute.

b) td
ANS: Element.
c) th
ANS: Element.
d) frame
ANS: Element.
e) name
ANS: Attribute.
f) select
ANS: Element.
g) type
ANS: Attribute.

27.5 What will the **frameset** produced by the following code look like? Assume that the pages imported are blank with white backgrounds and that the dimensions of the screen are 800 by 600. Sketch the layout, approximating the dimensions.

```
<frameset rows = "20%,*">
   <frame src = "hello.html" name = "hello" />
      <frameset cols = "150,*">
         <frame src = "nav.html" name = "nav" />
         <frame src = "deitel.html" name = "deitel" />
      </frameset>
</frameset>
```

ANS:



27.6 Write the XHTML markup to create a frame with a table of contents on the left side of the window, and have each entry in the table of contents use internal linking to scroll down the document frame to the appropriate subsection

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
```

```
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!-- Exercise 27.6 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.6</title>
   </head>
      <frameset cols = "175, *">
         <frame name = "sidebar" src = "sidebar.html" />
         <frame name = "main" src = "main.html" />
      </frameset>
</html>
```

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 27.6 Solution: sidebar.html -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.6</title>
   </head>
   <body>
         <ul>
            <li><a href = "main.html#camel" target = "main">
                   Camel Picture</a></li>
            <li><a href = "main.html#tictactoe"
                   target = "main">Tic Tac Toe</a></li>
            <li><a href = "main.html#table"
                   target = "main">Table Example</a></li>
         </ul>
   </body>
</html>
```

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 27.6 Solution: main.html-->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.6</title>
   </head>
   <body>
      <p>
         <strong>
            <a name = "camel">This is a picture of a
               camel:</a></br>
```

```
         </strong>
         <img src = "camel.gif" alt = "Camel picture" />
      </p>
      <p>
         <strong>
            <a name = "tictactoe">
               This is a tic tac toe example:</a>
         </strong>
      </p>
      <table border = "1">

         <!-- set all columns to be centered -->
         <colgroup>
            <col align = "center" />
            <col align = "center" />
            <col align = "center" />
         </colgroup>

         <!-- top column will span across 3 cells -->
         <tr>
            <th colspan = "3">
               <h1>This is the head of the
                  Tic-Tac-Toe table
               </h1>
            </th>
         </tr>

         <!-- row one of the table -->
         <tr>
            <td><h2>X</h2></td>
            <td><h2>O</h2></td>
            <td><h2>O</h2></td>
         </tr>

         <!-- row two of the table -->
         <tr>
            <td><h2>X</h2></td>
            <td><h2>X</h2></td>
            <td><h2>O</h2></td>
         </tr>

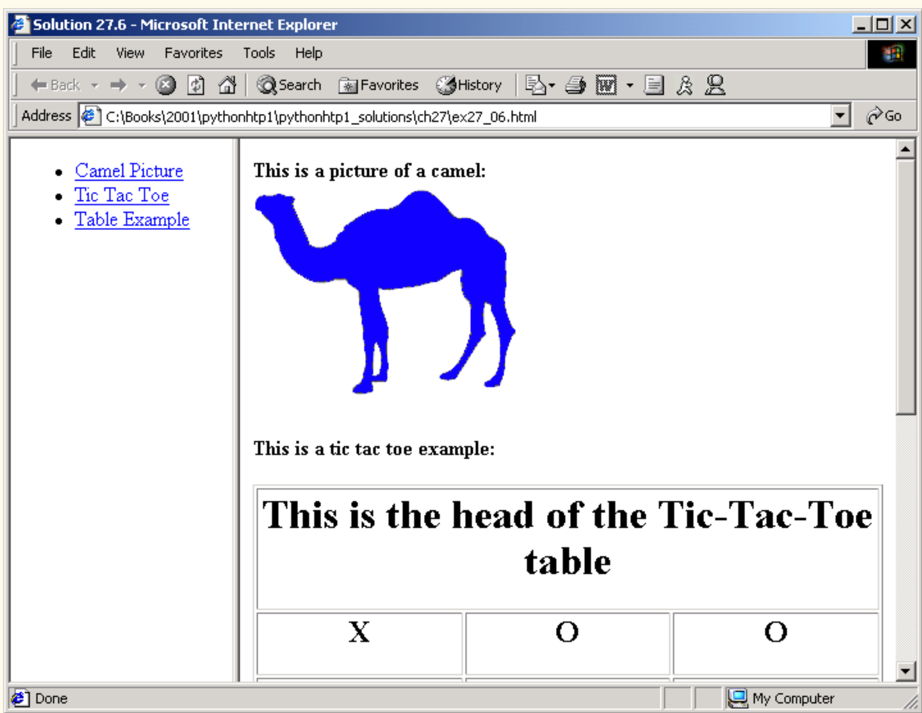
         <!-- row three of the table -->
         <tr>
            <td><h2>O</h2></td>
            <td><h2>O</h2></td>
            <td><h2>X</h2></td>
         </tr>
      </table>
      <p>
         <strong>
            <a name = "table">This is an example of a
               table</a>
         </strong>
      </p>
```

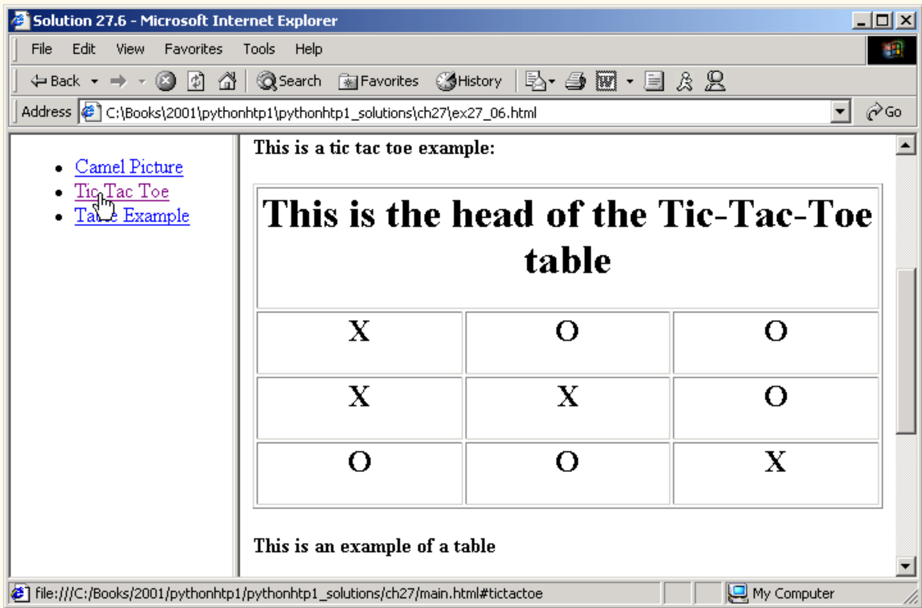
```
      <table border = "1" width = "40%">

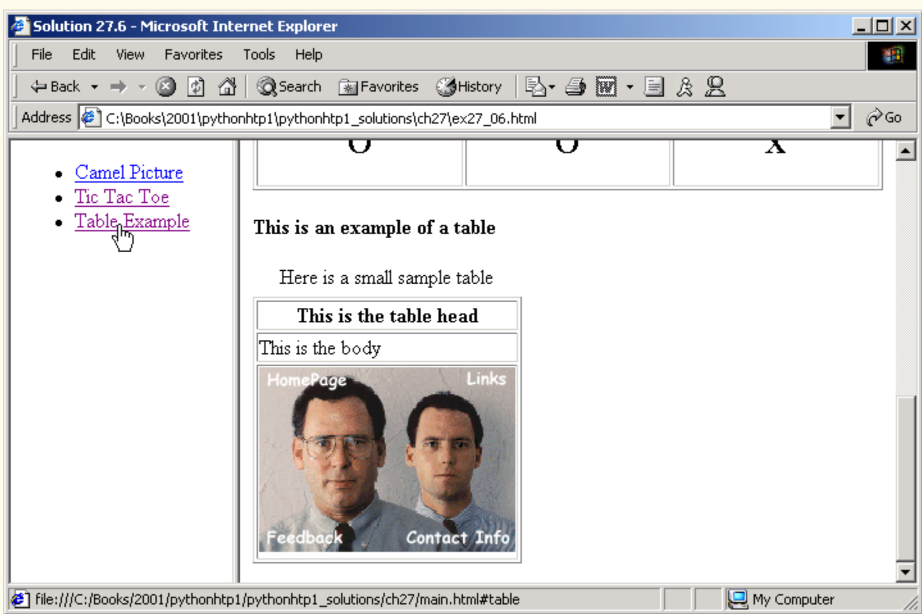
         <caption>Here is a small sample table</caption>

         <thead>
            <tr><th><strong>This is the table head</strong>
               </th></tr>
         </thead>

         <tbody>
            <tr>
               <td align = "left">This is the body
               </td>
            </tr>
            <tr>
               <td><img src = "deitel.gif" alt = "Deitel" />
               </td>
            </tr>
         </tbody>
      </table>
   </body>
</html>
```







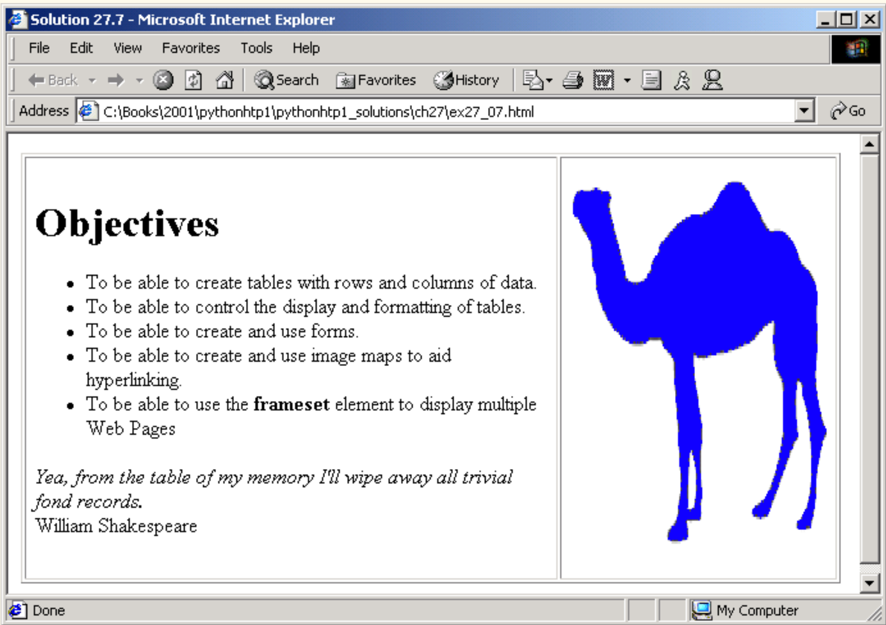
27.7 Create XHTML markup that produces the table shown in Fig. 27.12. Use **<em>** and **<strong>** tags as necessary. The image (**bug.jpg**) is included in the Chapter 27 examples directory on the CD-ROM that accompanies this book.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 27.7 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.7</title>
   </head>
   <body>
      <table border = "1" width = "640" cellpadding = "7">
         <tr>
            <td><h1>Objectives</h1>
               <ul>
                  <li>To be able to create tables with rows and
                     columns of data.</li>
                  <li>To be able to control the display and
                     formatting of tables.</li>
                  <li>To be able to create and use forms.</li>
                  <li>To be able to create and use image maps
                     to aid hyperlinking.</li>
                  <li>To be able to use the <strong>frameset
                     </strong> element to display multiple
                     Web Pages</li>
               </ul>
               <em>Yea, from the table of my memory I'll wipe
                  away all trivial fond records.</em><br />
                  William Shakespeare
            </td>
            <td><img src = "camel.gif" alt = "Camel picture"
               height = "310" width = "200" />
            </td>
         </tr>
      </table>
   </body>
</html>
```



27.8 Write an XHTML document that produces the table shown in Fig. 27.13.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 27.8 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.8</title>
   </head>
   <body>
      <h2>Table Example Page</h2>

      <!-- the table tag opens a new table -->
      <table border = "1" width = "40%">

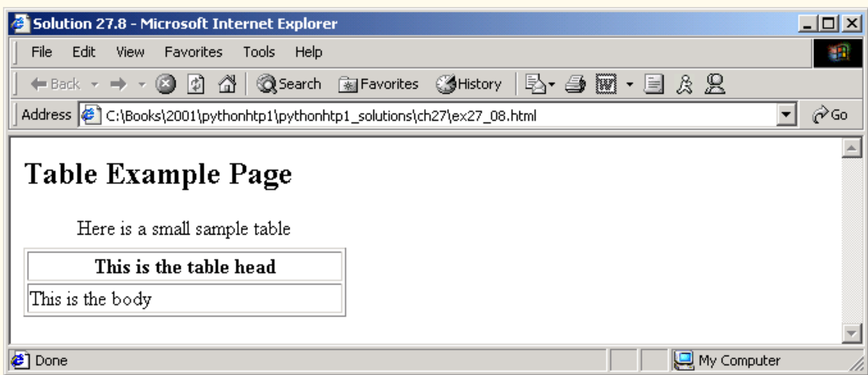
      <!-- use the caption tag to summarize the table's    -->
      <!-- contents (this helps for the visually impaired  -->
      <caption>Here is a small sample table</caption>

      <!-- the <thead> is the first horizontal              -->
      <!-- section. Use it to format the table header area.  -->
      <!-- <th> inserts a header cell and displays bold text -->
      <thead>
         <tr>
```

```
            <th>
               <strong>This is the table head</strong>
            </th>
         </tr>
      </thead>

      <!-- all of the main content goes in the <tbody> -->
      <!-- use this tag to format the entire section   -->
      <!-- <td> inserts a data cell, with regular text -->
      <tbody>
         <tr><td align = "left">This is the body</td></tr>
      </tbody>

      </table>
   </body>
</html>
```



27.9 A local university has asked you to create an XHTML document that allows potential students to provide feedback about their campus visits. Your XHTML document should contain a form with text boxes for names, addresses and e-mails. Provide check boxes that allow prospective students to indicate what they liked most about the campus. These check boxes should include: students, location, campus, atmosphere, dorm rooms and sports. Also, provide radio buttons that ask the prospective student how they became interested in the university. Options should include: friends, television, Internet and other. In addition, provide a text area for additional comments, a submit button and a reset button.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 27.9 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.9</title>
```

```
   </head>
   <body>
      <h1>College Visit Feedback Form</h1>
      <p>
         Please fill out this form to let us know how your
         visit was so that we can improve our facilities to
         better suit you and your peers' needs.
      </p>

      <form method = "post" action = "">

         <p>
            <input type = "hidden" name = "recipient"
               value = "college@.college.edu" />
            <input type = "hidden" name = "subject"
               value = "Visit Feedback" />
            <input type = "hidden" name = "redirect"
               value = "index.html" />
         </p>

         <!-- insert a textbox to gather information about -->
         <!-- the user                                     -->

         <p><label>Full Name:
            <input name = "fullname" type = "text"
               size = "40" />
         </label></p>

         <p><label>Address1:
            <input name = "address1" type = "text"
               size = "40" />
            </label></p>

         <p><label>Address2:
            <input name = "address2" type = "text"
               size = "40" />
            </label></p>

         <p><label>Zip Code:
            <input name = "zip" type = "text" size = "10" />
            </label></p>

         <p><label>E-mail:
            <input name = "email" type = "text" size = "25" />
            </label></p>

         <strong><em>Check off all of the characteristics that
            you enjoyed about the college:</em></strong><br />

         <!-- insert checkboxes for the user to check  -->
         <!-- off what he or she likes                 -->
         <p>
            <label>Campus
            <input name = "likes" type = "checkbox"
```

```
                  value = "campus" />
                  </label>

               <label>Students
               <input name = "likes" type = "checkbox"
                  value = "students" />
                  </label>

               <label>Location
               <input name = "likes" type = "checkbox"
                  value = "location" />
                  </label>
            </p>

            <p>
               <label>Atmosphere
               <input name = "likes" type = "checkbox"
                  value = "atmosphere" />
                  </label>

               <label>Dorm Rooms
               <input name = "likes" type = "checkbox"
                  value = "dormrooms" />
                  </label>

               <label>Sports
               <input name = "likes" type = "checkbox"
                  value = "sports" />
                  </label>
            </p>
               <strong><em>How did you become interested in
                  our college?</em></strong><br />

            <!-- radio buttons, user can only select one -->

            <p>
               <label>Friends
               <input name = "interest" type = "radio"
                  value = "friends" checked = "checked" />
                  </label>

               <label>TV Advertisement
               <input name = "interest" type = "radio"
                  value = "tv" />
                  </label>

               <label>Internet
               <input name = "interest" type = "radio"
                  value = "internet" />
                  </label>

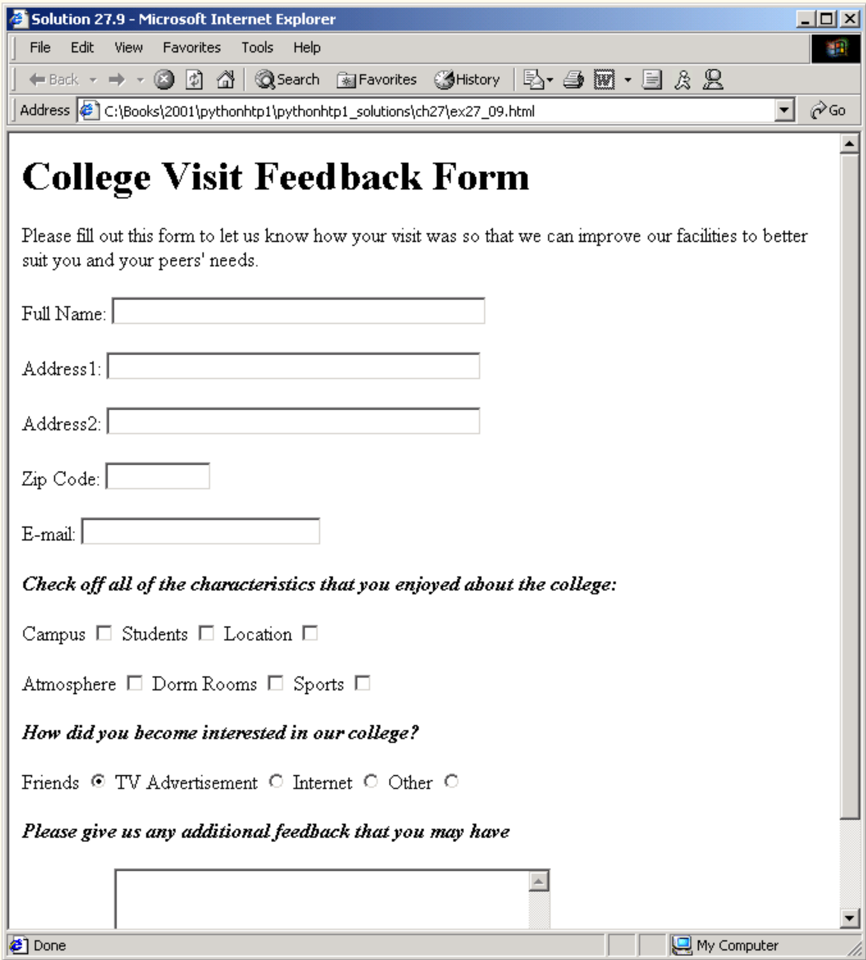
               <label>Other
               <input name = "interest" type = "radio"
                  value = "other" />
```

```
               </label>
         </p>

         <strong><em>Please give us any additional feedback
            that you may have</em></strong><br />

         <!-- user can enter in multiple lines of  -->
         <!-- information in a textarea            -->

         <p>
            <label>Comments:
               <textarea name = "comments" rows = "4"
                  cols = "40"></textarea>
               </label>
         </p>
         <p>
            <input type = "submit" value = "Submit" />
            <input type = "reset" value = "Clear" />
         </p>
      </form>
   </body>
</html>
```



27.10 Create an XHTML document titled "How to Get Good Grades." Use **<meta>** tags to include a series of keywords that describe your document.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 27.10 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.10</title>
```

```
      <meta name = "keywords" content = "grades, good, student,
         study, read, work, homework, " />
      <meta name = "description" content = "This web page will
         give you all the secrets to getting good grades at
         any level of education" />
   </head>
   <body>
      <h1>How to Get Good Grades</h1>
   </body>
</html>
```



27.11 Create an XHTML document that displays a tic-tac-toe table with player X winning. Use **<h2>** to mark up both Xs and Os. Center the letters in each cell horizontally. Title the game using an **<h1>** tag. This title should span all three columns. Set the table border to one.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 27.11 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 27.11</title>
   </head>
   <body>

      <table border = "1">

         <!-- set all columns to be centered -->
         <colgroup>
            <col align = "center" />
            <col align = "center" />
            <col align = "center" />
         </colgroup>

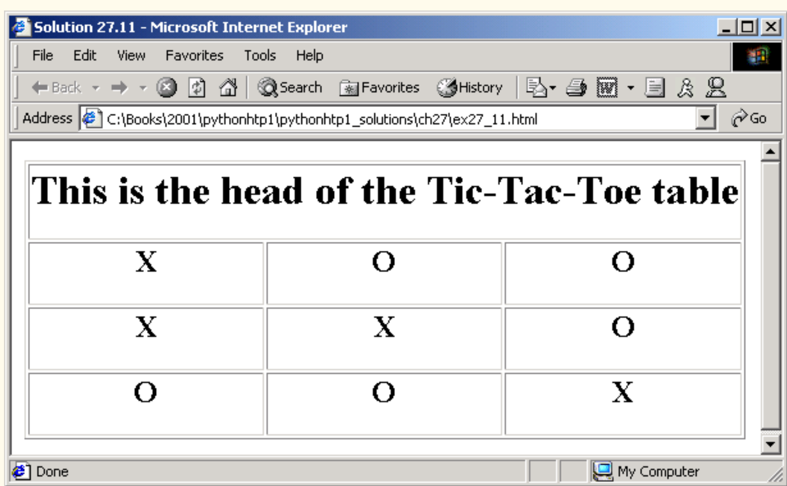
         <!-- top column will span across 3 cells -->
         <tr>
            <th colspan = "3">
```

```
               <h1>This is the head of the
                  Tic-Tac-Toe table
               </h1>
            </th>
         </tr>

         <!-- row one of the table -->
         <tr>
            <td><h2>X</h2></td>
            <td><h2>O</h2></td>
            <td><h2>O</h2></td>
         </tr>

          <!-- row two of the table -->
         <tr>
            <td><h2>X</h2></td>
            <td><h2>X</h2></td>
            <td><h2>O</h2></td>
         </tr>

          <!-- row three of the table -->
         <tr>
            <td><h2>O</h2></td>
            <td><h2>O</h2></td>
            <td><h2>X</h2></td>
         </tr>
      </table>
   </body>
</html>
```



[***Notes To Reviewers***]

- This chapter will be sent for second-round review.
- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send us e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **cheryl.yaeger@deitel.net**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
- Please be constructive. This book will be published soon. We all want to publish the best possible book.
- If you find something that is incorrect, please show us how to correct it.
- Please read all the back matter including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

U

V

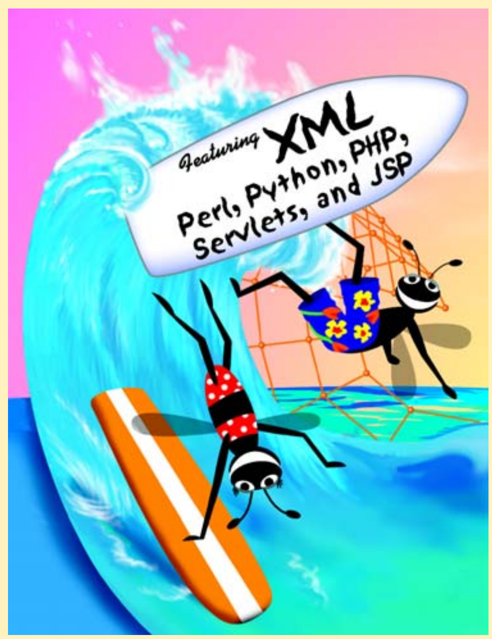
W

X

28

Cascading Style Sheets™ (CSS)

Objectives



- To take control of the appearance of a Web site by creating style sheets.
- To use a style sheet to give all the pages of a Web site the same look and feel.
- To use the **`class`** attribute to apply styles.
- To specify the precise font, size, color and other properties of displayed text.
- To specify element backgrounds and colors.
- To understand the box model and how to control the margins, borders and padding.
- To use style sheets to separate presentation from content.

Fashions fade, style is eternal.
Yves Saint Laurent

A style does not go out of style as long as it adapts itself to its period. When there is an incompatibility between the style and a certain state of mind, it is never the style that triumphs.
Coco Chanel

How liberating to work in the margins, outside a central perception.
Don DeLillo

I've gradually risen from lower-class background to lower-class foreground.
Marvin Cohen

Outline

28.1 Introduction

In Chapters 26 and 27, we introduced the Extensible Markup Language (XHTML) for marking up information. In this chapter, we shift our focus from marking up information to formatting and presenting information using a W3C technology called *Cascading Style Sheets* (*CSS*) that allows document authors to specify the presentation of elements on a Web page (spacing, margins, etc.) separately from the structure of the document (section headers, body text, links, etc.). This *separation of structure from presentation* simplifies maintaining and modifying a document's layout.

28.2 Inline Styles

A Web developer can declare document styles in many ways. In this section, we present *inline styles* that declare an individual element's format using *attribute* ***`style`***. Figure 28.1 applies inline styles to **`p`** elements to alter their font sizes and colors.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.1: inline.html -->
<!-- Using inline styles     -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Inline Styles</title>
   </head>

```

Fig. 28.1 Inline styles (part 1 of 2).

```
13     <body>
14
15        <p>This text does not have any style applied to it.</p>
16
17        <!-- The style attribute allows you to declare -->
18        <!-- inline styles. Separate multiple styles   -->
19        <!-- with a semicolon.                         -->
20        <p style = "font-size: 20pt">This text has the
21        <em>font-size</em> style applied to it, making it 20pt.
22        </p>
23
24        <p style = "font-size: 20pt; color: #0000ff">
25        This text has the <em>font-size</em> and
26        <em>color</em> styles applied to it, making it
27        20pt. and blue.</p>
28
29     </body>
30  </html>
```

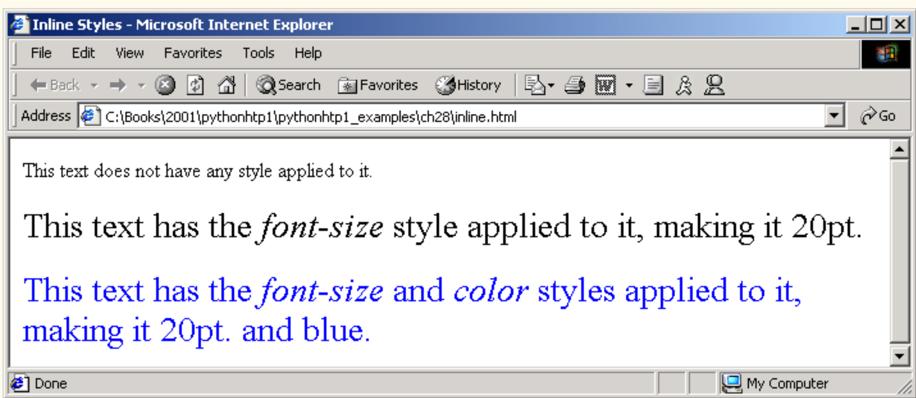


Fig. 28.1 Inline styles (part 2 of 2).

The first inline style declaration appears in line 20. Attribute **`style`** specifies the style for an element. Each *CSS property* (the ***`font-size`*** property in this case) is followed by a colon and a value. On line 20, we declare the **`p`** element to have 20-point text size. Line 21 uses element ***`em`*** to "emphasize" text, which most browsers do by making the font italic.

Line 24 specifies the two properties, **`font-size`** and ***`color`***, separated by a semicolon. In this line, we set the text's **`color`** to blue, using the hexadecimal code **`#0000ff`**. Color names may be used in place of hexadecimal codes, as we demonstrate in the next example. [*Note*: Inline styles override any other styles applied using the techniques we discuss later in this chapter.]

28.3 Embedded Style Sheets

In this section, we present a second technique for using style sheets called *embedded style sheets*. Embedded style sheets enable a Web-page author to embed an entire CSS docu-

ment in an XHTML document's **head** section. Figure 28.2 creates an embedded style sheet containing four styles.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.2: declared.html                        -->
<!-- Declaring a style sheet in the header section. -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Style Sheets</title>

      <!-- this begins the style sheet section -->
      <style type = "text/css">

         em       { background-color: #8000ff;
                    color: white }

         h1       { font-family: arial, sans-serif }

         p        { font-size: 14pt }

         .special { color: blue }

      </style>
   </head>

   <body>

      <!-- this class attribute applies the .blue style -->
      <h1 class = "special">Deitel & Associates, Inc.</h1>

      <p>Deitel & Associates, Inc. is an internationally
      recognized corporate training and publishing organization
      specializing in programming languages, Internet/World
      Wide Web technology and object technology education.
      Deitel & Associates, Inc. is a member of the World Wide
      Web Consortium. The company provides courses on Java,
      C++, Visual Basic, C, Internet and World Wide Web
      programming, and Object Technology.</p>

      <h1>Clients</h1>
      <p class = "special"> The company's clients include many
      <em>Fortune 1000 companies</em>, government agencies,
      branches of the military and business organizations.
      Through its publishing partnership with Prentice Hall,
      Deitel & Associates, Inc. publishes leading-edge
      programming textbooks, professional books, interactive
      CD-ROM-based multimedia Cyber Classrooms, satellite
      courses and World Wide Web courses.</p>
```

Fig. 28.2 Declaring styles in the **head** of a document (part 1 of 2).

```
50
51       </body>
52    </html>
```

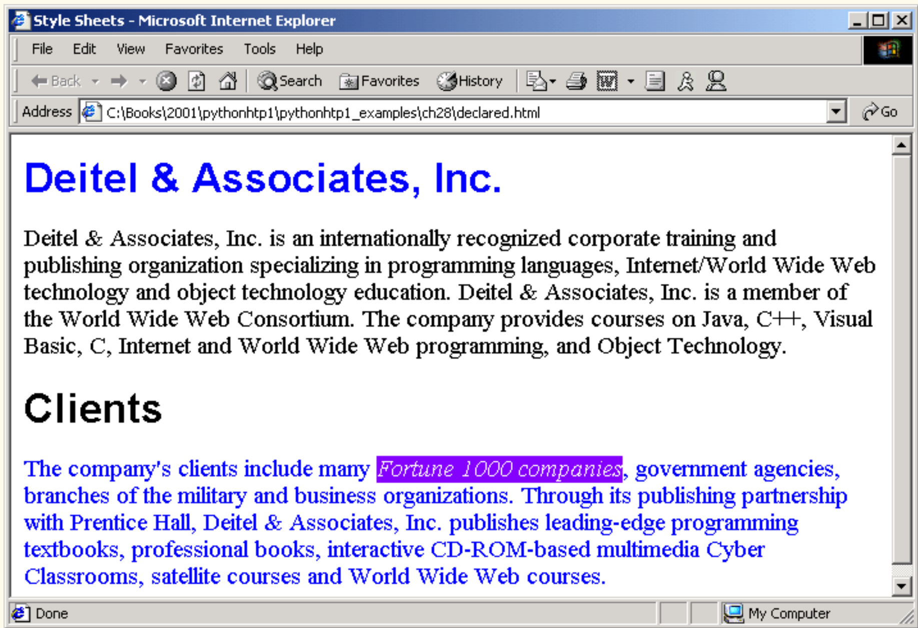


Fig. 28.2 Declaring styles in the **head** of a document (part 2 of 2).

The **style** element (lines 13–24) defines the embedded style sheet. Styles placed in the **head** apply to matching elements in the entire document, not just to a single element. The **type** attribute specifies the *Multipurpose Internet Mail Extension* (*MIME*) *type* that describes a file's content. CSS documents use the MIME type **text/css**. Other MIME types include **image/gif** (for GIF images) and **text/javascript** (for the JavaScript scripting language).

The body of the style sheet (lines 15–22) declares the *CSS rules* for the style sheet. We declare rules for **em** (lines 15–16), **h1** (line 18) and **p** (line 20) elements. When the browser renders this document, it applies the properties defined in these rules to each element to which the rule applies. For example, the rule on lines 15–16 will be applied to all **em** elements. The body of each rule is enclosed in curly braces (**{** and **}**). We declare a *style class* named **special** in line 22. Class declarations are preceded with a period and are applied to elements only of that class. We discuss how to apply a style class momentarily.

CSS rules that embedded style sheets use the same syntax as inline styles; the property name is followed by a colon (**:**) and the value of that property. Multiple properties are separated by *semicolons* (**;**). In this example, the **color** property specifies the color of text in an element line and property ***background-color*** specifies the background color of the element.

The **font-family** property (line 18) specifies the name of the font to use. In this case, we use the **arial** font. The second value, **sans-serif**, is a *generic font family*.

Not all users have the same fonts installed on their computers, so Web-page authors often specify a comma-separated list of fonts to use for a particular style. The browser attempts to use the fonts in the order they appear in the list. Many Web-page authors end a font list with a generic font family name in case the other fonts are not installed on the user's computer. In this example, if the **arial** font is not found on the system, the browser instead displays a generic **sans-serif** font such as **helvetica** or **verdana**. Other generic font families include **serif** (e.g., **times new roman**, **Georgia**), **cursive** (e.g., **script**), **fantasy** (e.g., **critter**) and **monospace** (e.g., **courier**, **fixedsys**).

The ***font-size*** *property* (line 20) specifies a 14-point font. Other possible measurements, in addition to ***pt*** (point), are introduced later in the chapter. Relative values— ***xx-small***, ***x-small***, ***small***, ***smaller***, ***medium***, ***large***, ***larger***, ***x-large*** and ***xx-large*** also can be used. Generally, relative values for **font-size** are preferred over point sizes because an author does not know the specific measurements of the display for each client. For example, a user may wish to view a Web page on a handheld device with a small screen. Specifying an 18-point font size in a style sheet prevents such a user from seeing more than one or two characters at a time. However, if a relative font size is specified, such as **large** or **larger**, the actual size is determined by the browser that displays the font.

Line 30 uses attribute ***class*** in an **h1** element to apply a *style class*—in this case class **special** (declared as **.special** in the style sheet). When the browser renders the **h1** element, notice that the text appears on screen with both the properties of an **h1** element (**arial** or **sans-serif** font defined at line 18) and the properties of the **.special** style class applied (the color **blue** defined on line 22).

The **p** element and the **.special** class style are applied to the text in lines 42–49. All styles applied to an element (the *parent*, or *ancestor*, *element*) also apply to that element's nested elements (*descendant elements*). The **em** element *inherits* the style from the **p** element (namely, the 14-point font size in line 20), but retains its italic style. However, this property overrides the **color** property of the **special** class because the **em** element has its own **color** property. We discuss the rules for resolving these conflicts in the next section.

28.4 Conflicting Styles

Cascading style sheets are "cascading" because styles may be defined by a user, an author or a *user agent* (e.g., a Web browser). Styles defined by authors take precedence over styles defined by the user, and styles defined by the user take precedence over styles defined by the user agent. Styles defined for parent and ancestor elements are also inherited by child and descendant elements. In this section, we discuss the rules for resolving conflicts between styles defined for elements and styles inherited from parent and ancestor elements.

Figure 28.2 presented an example of *inheritance* in which a child **em** element inherited the **font-size** property from its parent **p** element. However, in Fig. 28.2, the child **em** element had a **color** property that conflicted with (i.e., had a different value than) the **color** property of its parent **p** element. Properties defined for child and descendant elements have a greater *specificity* than properties defined for parent and ancestor elements. According to the W3C CSS Recommendation, conflicts are resolved in favor of properties with a higher specificity. In other words, the styles defined for the child (or descendant) are more specific than the styles for that child's parent (or ancestor) element; therefore, the

child's styles take precedence. Figure 28.3 illustrates examples of inheritance and specificity.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig 28.3: advanced.html    -->
<!-- More advanced style sheets -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>More Styles</title>

      <style type = "text/css">

         a.nodec  { text-decoration: none }

         a:hover  { text-decoration: underline;
                    color: red;
                    background-color: #ccffcc }

         li em    { color: red;
                    font-weight: bold }

         ul       { margin-left: 75px }

         ul ul    { text-decoration: underline;
                    margin-left: 15px }

      </style>
   </head>

   <body>

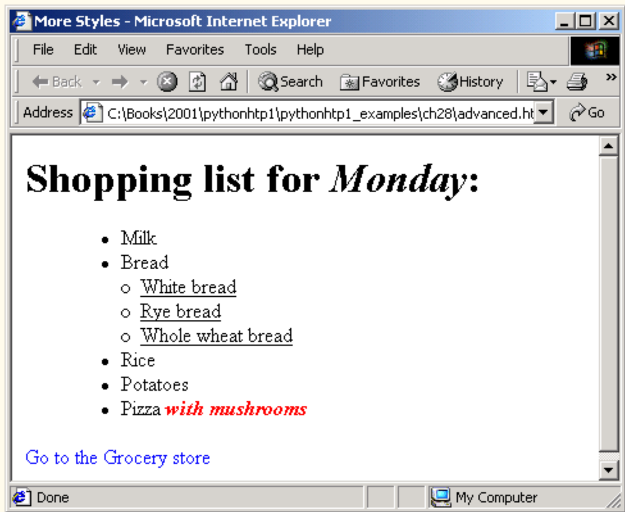
      <h1>Shopping list for <em>Monday</em>:</h1>

      <ul>
         <li>Milk</li>
         <li>Bread
            <ul>
               <li>White bread</li>
               <li>Rye bread</li>
               <li>Whole wheat bread</li>
            </ul>
         </li>
         <li>Rice</li>
         <li>Potatoes</li>
         <li>Pizza <em>with mushrooms</em></li>
      </ul>

      <p><a class = "nodec" href = "http://www.food.com">
```

Fig. 28.3 Inheritance in style sheets (part 1 of 2).

```
50          Go to the Grocery store</a></p>
51
52       </body>
53    </html>
```



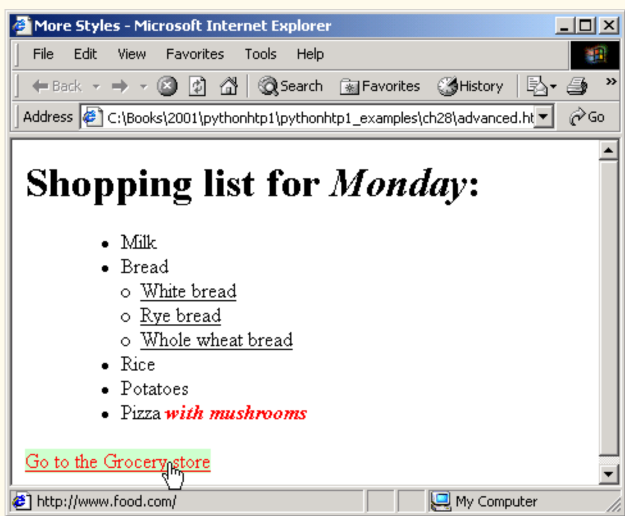


Fig. 28.3 Inheritance in style sheets (part 2 of 2).

Line 14 applies property **`text-decoration`** to all **`a`** elements whose **`class`** attribute is set to **`nodec`**. The **`text-decoration`** property applies *decorations* to text within an element. By default, browsers underline the text marked up with an **`a`** element. Here, we set the **`text-decoration`** property to **`none`** to indicate that the browser should not underline hyperlinks. Other possible values for **`text-decoration`** include ***`blink`***, ***`overline`***,

`line-through` and ***`underline`***. The **`.nodec`** appended to **`a`** is an extension of class styles; this style applies only to **`a`** elements that specify **`nodec`** as their class.

Lines 16–18 specify a style for **`hover`**, which is a *pseudoclass*. Pseudoclasses give the author access to content not specifically declared in the document. The **`hover`** pseudoclass is activated dynamically when the user moves the mouse cursor over an element.

Portability Tip 28.1

To ensure that your style sheets work in various Web browsers, test your style sheets on all client Web browsers that will render documents using your styles.

Lines 20–21 declare a style for all **`em`** elements that are descendants of **`li`** elements. In the screen output of Fig. 28.3, notice that **Monday** (which line 33 contains in an **`em`** element) does not appear in bold red, because the **`em`** element is not in an **`li`** element. However, the **`em`** element containing **with mushrooms** (line 46) is in an **`li`** element; therefore, it is formatted in bold red.

The syntax for applying rules to multiple elements is similar. For example, to apply the rule in lines 20–21 to all **`li`** and **`em`** elements, you would separate the elements with commas, as follows:

```
li, em  { color: red;
          font-weight: bold }
```

Lines 25–26 specify that all nested lists (**`ul`** elements that are descendants of **`ul`** elements) be underlined and have a left-hand margin of 15 pixels. A pixel is a *relative-length measurement*—it varies in size, based on screen resolution. Other relative lengths are ***`em`*** (the so-called "*M*-height" of the font, which is usually set to the height of an uppercase *M*), ***`ex`*** (the so-called "*x*-height" of the font, which is usually set to the height of a lowercase *x*) and percentages (e.g., **`margin-left: 10%`**). To set an element to display text at 150% of its default text size, the author could use the syntax

```
font-size: 1.5em
```

Other units of measurement available in CSS are *absolute-length measurements*—i.e., units that do not vary in size based on the system. These units are ***`in`*** (inches), ***`cm`*** (centimeters), ***`mm`*** (millimeters), **`pt`** (points; 1 **`pt`**=1/72 **`in`**) and ***`pc`*** (picas—1 **`pc`** = 12 **`pt`**).

Good Programming Practice 28.1

Whenever possible, use relative-length measurements. If you use absolute-length measurements, your document may not be readable on some client browsers (e.g., wireless phones).

In Fig. 28.3, the entire list is indented because of the 75-pixel left-hand margin for top-level **`ul`** elements. However, the nested list is indented only 15 pixels more (not another 75 pixels) because the child **`ul`** element's **`margin-left`** property overrides the parent **`ul`** element's **`margin-left`** property.

28.5 Linking External Style Sheets

Style sheets are a convenient way to create a document with a uniform theme. With *external style sheets* (i.e., separate documents that contain only CSS rules), Web-page authors can provide a uniform look and feel to an entire Web site. Different pages on a site can all use the same style sheet. Then, when changes to the style are required, the Web-page author needs to modify only a single CSS file to make style changes across the entire Web site.

Figure 28.4 presents an external style sheet and Fig. 28.5 contains an XHTML document that references the style sheet.

```
/* Fig. 28.4: styles.css  */
/* An external stylesheet */

a        { text-decoration: none }

a:hover  { text-decoration: underline;
           color: red;
           background-color: #ccffcc }

li em    { color: red;
           font-weight: bold;
           background-color: #ffffff }

ul       { margin-left: 2cm }

ul ul    { text-decoration: underline;
           margin-left: .5cm }
```

Fig. 28.4 External style sheet (**styles.css**).

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.5: external.html       -->
<!-- Linking external style sheets  -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Linking External Style Sheets</title>
      <link rel = "stylesheet" type = "text/css"
         href = "styles.css" />
   </head>

   <body>

      <h1>Shopping list for <em>Monday</em>:</h1>
      <ul>
         <li>Milk</li>
         <li>Bread
            <ul>
               <li>White bread</li>
               <li>Rye bread</li>
               <li>Whole wheat bread</li>
            </ul>
         </li>
         <li>Rice</li>
         <li>Potatoes</li>
```

Fig. 28.5 Linking an external style sheet (part 1 of 2).

```
29             <li>Pizza <em>with mushrooms</em></li>
30          </ul>
31
32          <p>
33          <a href = "http://www.food.com">Go to the Grocery store</a>
34          </p>
35
36       </body>
37    </html>
```



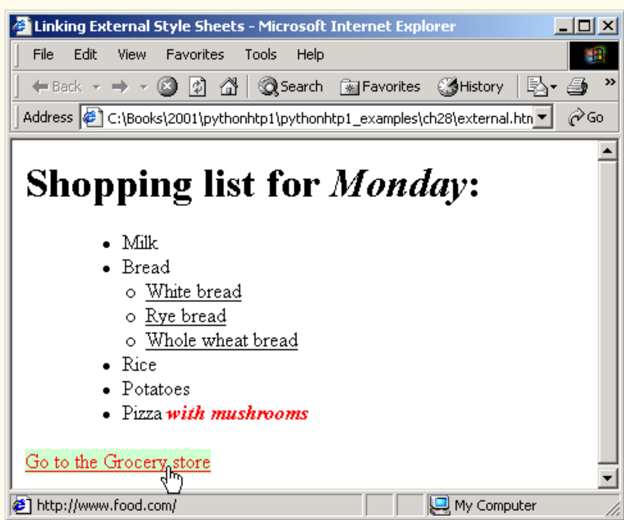


Fig. 28.5 Linking an external style sheet (part 2 of 2).

Lines 11–12 (Fig. 28.5) show a **`link`** *element*, which uses the **`rel`** *attribute* to specify a *relationship* between the current document and another document. In this case, we declare

the linked document to be a ***`stylesheet`*** for this document. The **`type`** attribute specifies the MIME type as **`text/css`**. The **`href`** attribute provides the URL for the document containing the style sheet .

Software Engineering Observation 28.1

Style sheets are reusable. Creating them once and reusing them reduces programming effort.

Software Engineering Observation 28.2

*The **`link`** element can be placed only in the **`head`** element. The user can specify **`next`** and **`previous`**, which allows the user to link a whole series of documents. This feature allows browsers to print a large collection of related documents at once.* (*In Internet Explorer, select* ***Print all linked documents*** *in the* ***Print...*** *submenu of the* ***File*** *menu.*)

28.6 W3C CSS Validation Service

The W3C provides a validation service (**`jigsaw.w3.org/css-validator`**) that validates external CSS documents to ensure that they conform to the W3C CSS Recommendation. Like XHTML validation, CSS validation ensures that style sheets have correct syntax. The validator provides the option of either entering the CSS document's URL, pasting the CSS document's contents into a text area or uploading a CSS document from disk. Figure 28.6 illustrates uploading a CSS document from a disk.

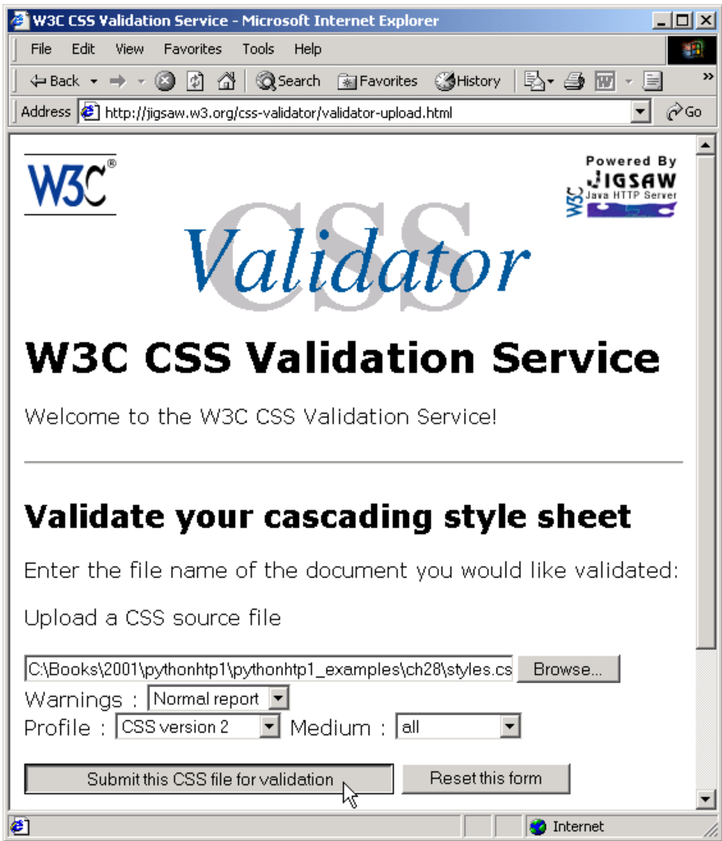


Fig. 28.6 Validating a CSS document.

Figure 28.7 shows the results of validating **`styles.css`** (Fig. 28.4), using the file upload feature available at

`jigsaw.w3.org/css-validator/validator-upload.html`

To validate the document, click the **Browse** button to locate the file on your computer. After locating the file, click **Submit this CSS file for validation** to upload the file for validation. [*Note*: Like many W3C technologies, CSS is being developed in stages (or *versions*). The current version under development is Version 3.]

28.7 Positioning Elements

Prior to CSS, controlling the positioning of elements in an XHTML document was difficult—the browser determined positioning. CSS introduces the ***`position`*** property and a

capability called *absolute positioning*, which provides authors greater control over how document elements are displayed. Figure 28.8 demonstrates absolute positioning.

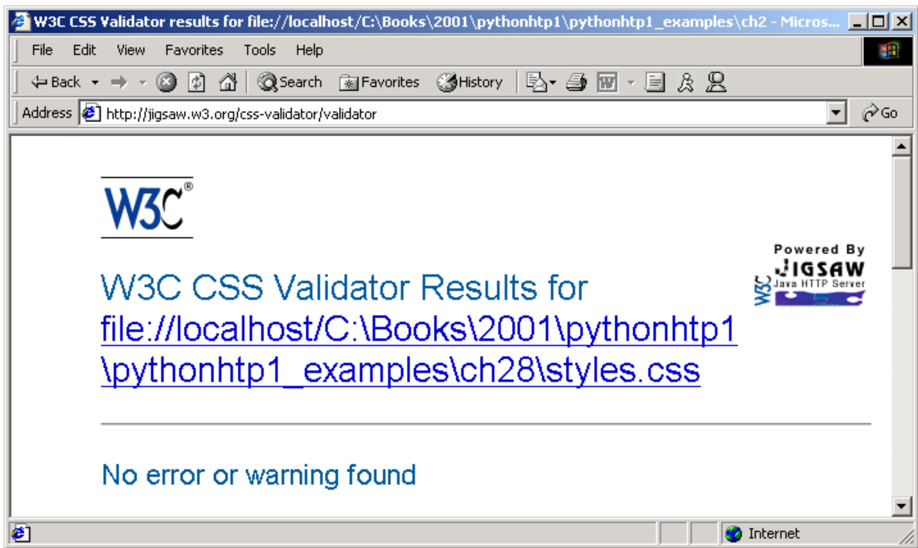


Fig. 28.7 CSS validation results. (Courtesy of World Wide Web Consortium (W3C).)

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig 28.8: positioning.html       -->
<!-- Absolute positioning of elements -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Absolute Positioning</title>
   </head>

   <body>

      <p><img src = "i.gif" style = "position: absolute;
         top: 0px; left: 0px; z-index: 1"
         alt = "First positioned image" /></p>
      <p style = "position: absolute; top: 50px; left: 50px;
         z-index: 3; font-size: 20pt;">Positioned Text</p>
      <p><img src = "circle.gif" style = "position: absolute;
         top: 25px; left: 100px; z-index: 2" alt =
         "Second positioned image" /></p>

   </body>
```

Fig. 28.8 Positioning elements with CSS (part 1 of 2).

```
25   </html>
```

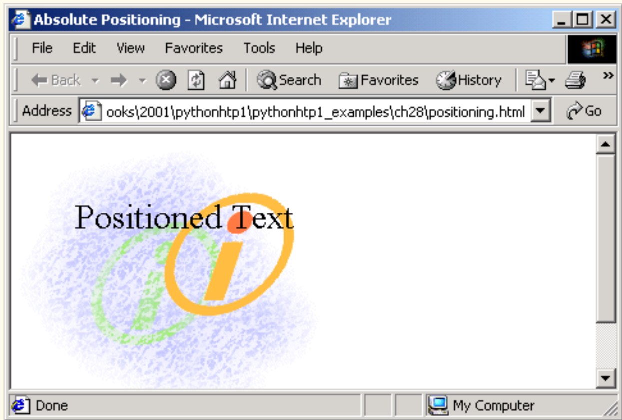


Fig. 28.8 Positioning elements with CSS (part 2 of 2).

Lines 15–17 position the first **img** element (**i.gif**) on the page. Specifying an element's **position** as ***absolute*** removes the element from the normal flow of elements on the page, instead positioning the element according to the distance from the **top**, **left**, **right** or **bottom** margins of its *containing block* (i.e., an element such as **body** or **p**). Here, we position the element to be **0** pixels away from both the **top** and **left** margins of the **body** element.

The ***z-index*** attribute allows you to layer overlapping elements properly. Elements that have higher **z-index** values are displayed in front of elements with lower **z-index** values. In this example, **i.gif** has the lowest **z-index** (**1**), so it displays in the background. The **img** element at lines 20–22 (**circle.gif**) has a **z-index** of **2**, so it displays in front of **i.gif**. The **p** element at lines 18–19 (**Positioned Text**) has a **z-index** of **3**, so it displays in front of the other two. If you do not specify a **z-index** or if elements have the same **z-index** value, the elements are placed from background to foreground in the order they are encountered in the document.

Absolute positioning is not the only way to specify page layout. Figure 28.9 demonstrates *relative positioning* in which elements are positioned relative to other elements.

Setting the **position** property to **relative**, as in class **super** (lines 21–22), lays out the element on the page and offsets the element by the specified **top**, **bottom**, **left** or **right** values. Unlike absolute positioning, relative positioning keeps elements in the general flow of elements on the page, so positioning is relative to other elements in the flow.

```
1   <?xml version = "1.0"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5   <!-- Fig. 28.9: positioning2.html       -->
6   <!-- Relative positioning of elements  -->
```

Fig. 28.9 Relative positioning of elements (part 1 of 3).

```

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Relative Positioning</title>

      <style type = "text/css">

         p           { font-size: 1.3em;
                       font-family: verdana, arial, sans-serif }

         span        { color: red;
                       font-size: .6em;
                       height: 1em }

         .super      { position: relative;
                       top: -1ex }

         .sub        { position: relative;
                       bottom: -1ex }

         .shiftleft  { position: relative;
                       left: -1ex }

         .shiftright { position: relative;
                       right: -1ex }

      </style>
   </head>

   <body>

      <p>The text at the end of this sentence
      <span class = "super">is in superscript</span>.</p>

      <p>The text at the end of this sentence
      <span class = "sub">is in subscript</span>.</p>

      <p>The text at the end of this sentence
      <span class = "shiftleft">is shifted left</span>.</p>

      <p>The text at the end of this sentence
      <span class = "shiftright">is shifted right</span>.</p>

   </body>
</html>
```

Fig. 28.9 Relative positioning of elements (part 2 of 3).

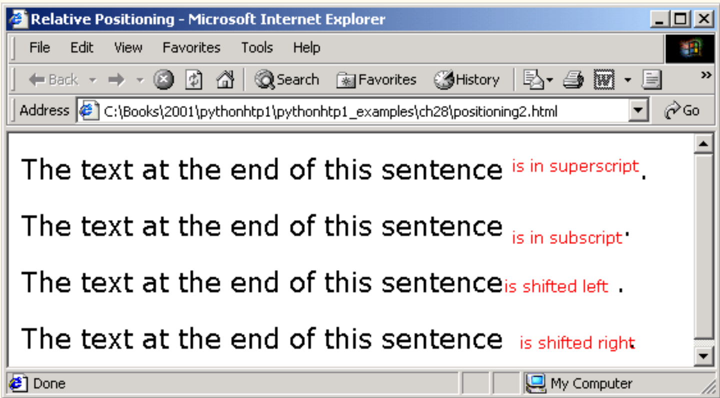


Fig. 28.9 Relative positioning of elements (part 3 of 3).

We introduce the ***span*** element in line 39. Element **span** is a *grouping element*—it does not apply any inherent formatting to its contents. Its primary purpose is to apply CSS rules or ***id*** *attributes* to a block of text. Element **span** is an *inline-level element*—it is displayed inline with other text and with no line breaks. Lines 17–19 define the CSS rule for **span**. A similar element is the ***div*** element, which also applies no inherent styles but is displayed on its own line, with margins above and below (a *block-level element*).

Common Programming Error 28.1

Because relative positioning keeps elements in the flow of text in your documents, be careful to avoid unintentionally overlapping text.

28.8 Backgrounds

CSS also provides control over the element backgrounds. In previous examples, we introduced the **background-color** property. CSS also can add background images to documents. Figure 28.10 adds a corporate logo to the bottom-right corner of the document. This logo stays fixed in the corner, even when the user scrolls up or down the screen.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.10: background.html              -->
<!-- Adding background images and indentation -->

<html xmlns = "http://www.w3 .org/1999/xhtml">
   <head>
      <title>Background Images</title>

      <style type = "text/css">

         body  { background-image: url(logo.gif);
```

Fig. 28.10 Adding a background image with CSS (part 1 of 2).

```
15                       background-position: bottom right;
16                       background-repeat: no-repeat;
17                       background-attachment: fixed; }
18
19               p      { font-size: 18pt;
20                        color: #aa5588;
21                        text-indent: 1em;
22                        font-family: arial, sans-serif; }
23
24               .dark { font-weight: bold; }
25
26            </style>
27         </head>
28
29         <body>
30
31            <p>
32            This example uses the background-image,
33            background-position and background-attachment
34            styles to place the <span class = "dark">Deitel
35            & Associates, Inc.</span> logo in the bottom,
36            right corner of the page. Notice how the logo
37            stays in the proper position when you resize the
38            browser window.
39            </p>
40
41         </body>
42      </html>
```

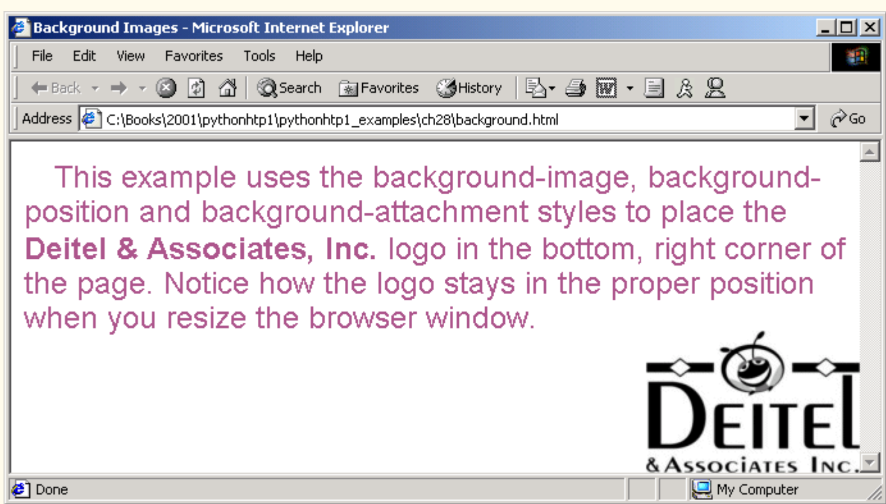


Fig. 28.10 Adding a background image with CSS (part 2 of 2).

The ***background-image*** *property* (line 14) specifies the image URL for the image **logo.gif** in the format **url(***fileLocation***)**. The Web-page author can set the **background-color** in case the image is not found.

The ***background-position*** *property* (line 15) places the image on the page. The keywords ***top***, ***bottom***, ***center***, ***left*** and ***right*** are used individually or in combination for vertical and horizontal positioning. Image can be positioned using lengths by specifying the horizontal length followed by the vertical length. For example, to position the image as vertically centered (positioned at 50% of the distance across the screen) and 30 pixels from the top, use

```
background-position: 50% 30px;
```

The ***background-repeat*** *property* (line 16) controls the *tiling* of the background image. Tiling places multiple copies of the image next to each other to fill the background. Here, we set the tiling to ***no-repeat*** to display only one copy of the background image. The **background-repeat** property can be set to ***repeat*** (the default) to tile the image vertically and horizontally, ***repeat-x*** to tile the image only horizontally or ***repeat-y*** to tile the image only vertically.

The final property setting, ***background-attachment: fixed*** (line 17), fixes the image in the position specified by **background-position**. Scrolling the browser window does not move the image from its position. The default value, ***scroll***, moves the image as the user scrolls through the document.

Line 21 indents the first line of text in the element by the specified amount, in this case **1em**. An author might use this property to create a Web page that reads more like a novel, in which the first line of every paragraph is indented.

Line 24 uses the ***font-weight*** *property* to specify the "boldness" of text. Possible values are ***bold***, ***normal*** (the default), ***bolder*** (bolder than **bold** text) and ***lighter*** (lighter than **normal** text). Boldness also can be specified with multiples of 100, from 100 to 900 (e.g., **100**, **200**, …, **900**). Text specified as **normal** is equivalent to **400**, and **bold** text is equivalent to **700**. However, many systems do not have fonts that scale this finely, so using the values from **100** to **900** might not display the desired effect.

Another CSS property that formats text is the ***font-style*** *property*, which allows the developer to set text to ***none***, ***italic*** or ***oblique*** (**oblique** defaults to **italic** if the system does not support oblique text).

28.9 Element Dimensions

In addition to positioning elements, CSS rules can specify the actual dimensions of each page element. Figure 28.11 demonstrates how to set the dimensions of elements.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.11: width.html                    -->
<!-- Setting box dimensions and aligning text  -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Box Dimensions</title>
```

Fig. 28.11 Setting box dimensions and aligning text (part 1 of 2).

```

      <style type = "text/css">

         div { background-color: #ffccff;
               margin-bottom: .5em }
      </style>

   </head>

   <body>

      <div style = "width: 20%">Here is some
      text that goes in a box which is
      set to stretch across twenty percent
      of the width of the screen.</div>

      <div style = "width: 80%; text-align: center">
      Here is some CENTERED text that goes in a box
      which is set to stretch across eighty percent of
      the width of the screen.</div>

      <div style = "width: 20%; height: 30%; overflow: scroll">
      This box is only twenty percent of
      the width and thirty percent of the height.
      What do we do if it overflows? Set the
      overflow property to scroll!</div>

   </body>
</html>
```

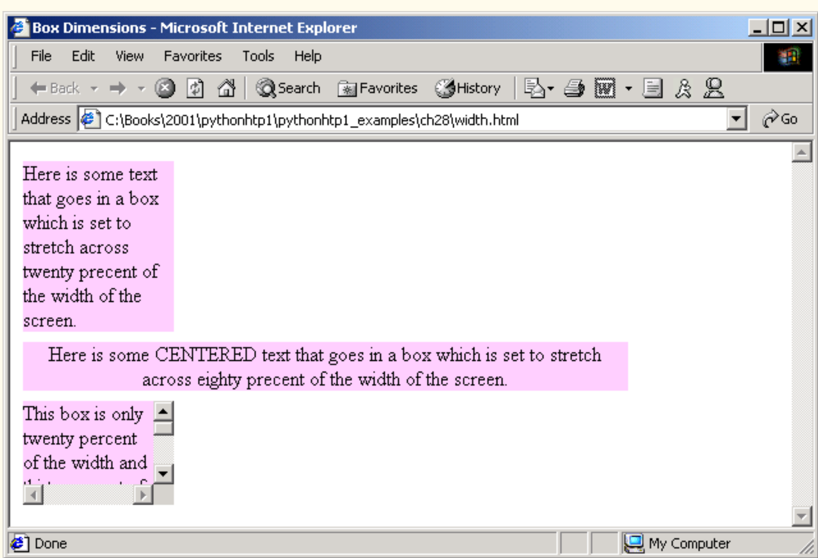


Fig. 28.11 Setting box dimensions and aligning text (part 2 of 2).

The inline style in line 22 illustrates how to set the ***width*** of an element on screen; here, we indicate that the **div** element should occupy 20% of the screen width. Most elements are left-aligned by default; however, this alignment can be altered to position the element elsewhere. The height of an element can be set similarly, using the ***height*** property. The **height** and **width** values also can be assigned relative and absolute lengths. For example

```
width: 10em
```

sets the element's width to be equal to 10 times the font size. Line 27 sets text in the element to be ***center*** aligned; some other values for the **text-align** property are ***left*** and ***right***.

One problem with setting both dimensions of an element is that the content inside the element can exceed the set boundaries, in which case the element is simply made large enough for all the content to fit. However, in line 32, we set the ***overflow*** property to ***scroll***, a setting that adds scrollbars if the text overflows the boundaries.

28.10 Text Flow and the Box Model

A browser normally places text and elements on screen in the order in which they appear in the XHTML document. However, as we have seen with absolute positioning, it is possible to remove elements from the normal flow of text. *Floating* allows you to move an element to one side of the screen; other content in the document then flows around the floated element. In addition, each block-level element has a box drawn around it, known as the *box model*. The properties of this box can be adjusted to control the amount of padding inside the element and the margins outside the element (Fig. 28.12).

In addition to text, whole elements can be *floated* to the left or right of content. This means that any nearby text wraps around the floated element. For example, in lines 30–32 we float a **div** element to the **right** side of the screen. As you can see from the sample screen capture, the text from lines 34–41 flows cleanly to the left and underneath the **div** element.

The second property on line 30, **margin**, specifies the distance between the edge of the element and any other element on the page. When the browser renders elements using the box model, the content of each element is surrounded by *padding*, a *border* and a *margin* (Fig. 6.13).

Margins for individual sides of an element can be specified by using ***margin-top***, ***margin-right***, ***margin-left*** and ***margin-bottom***.

Lines 43–45 specify a **div** element that floats at the right side of the content. Property ***padding*** for the **div** element is set to **.5em**. *Padding* is the distance between the content inside an element and the element's border. Like the **margin**, the **padding** can be set for each side of the box, with ***padding-top***, ***padding-right***, ***padding-left*** and ***padding-bottom***.

A portion of lines 54–55 show that you can interrupt the flow of text around a **float**ed element by setting the **clear** property to the same direction as that in which the element is **float**ed—**right** or **left**. Setting the ***clear*** property to ***all*** interrupts the flow on both sides of the document.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.12: floating.html           -->
<!-- Floating elements and element boxes -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Flowing Text Around Floating Elements</title>

      <style type = "text/css">

         div { background-color: #ffccff;
               margin-bottom: .5em;
               font-size: 1.5em;
               width: 50% }

         p   { text-align: justify; }

      </style>

   </head>

   <body>

      <div style = "text-align: center">
         Deitel & Associates, Inc.</div>

      <div style = "float: right; margin: .5em;
         text-align: right">
         Corporate Training and Publishing</div>

      <p>Deitel & Associates, Inc. is an internationally
      recognized corporate training and publishing organization
      specializing in programming languages, Internet/World
      Wide Web technology and object technology education.
      Deitel & Associates, Inc. is a member of the World Wide
      Web Consortium. The company provides courses on Java,
      C++, Visual Basic, C, Internet and World Wide Web
      programming, and Object Technology.</p>

      <div style = "float: right; padding: .5em;
         text-align: right">
         Leading-edge Programming Textbooks</div>

      <p>The company's clients include many Fortune 1000
      companies, government agencies, branches of the military
      and business organizations. Through its publishing
      partnership with Prentice Hall, Deitel & Associates,
      Inc. publishes leading-edge programming textbooks,
      professional books, interactive CD-ROM-based multimedia
```

Fig. 28.12 Floating elements, aligning text and setting box dimensions (part 1 of 2).

```
            Cyber Classrooms, satellite courses and World Wide Web
            courses.<span style = "clear: right"> Here is some
            unflowing text. Here is some unflowing text.</span></p>

      </body>
   </html>
```

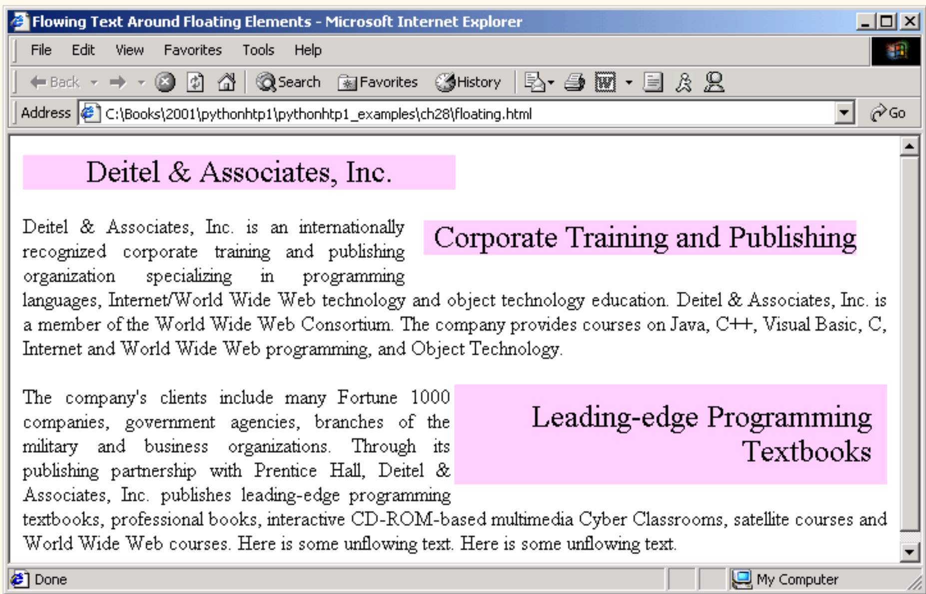


Fig. 28.12 Floating elements, aligning text and setting box dimensions (part 2 of 2).

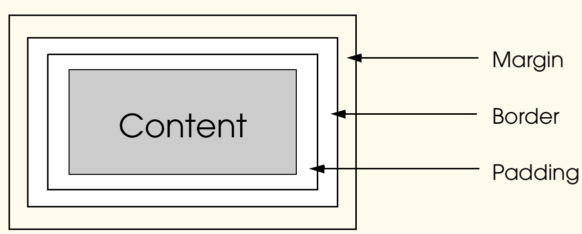


Fig. 28.13 Box model for block-level elements.

Another property of every block-level element on screen is the border, which lies between the padding space and the margin space and has numerous properties for adjusting its appearance as shown in Fig. 28.14.

```
<?xml version = "1.0"?>
```

Fig. 28.14 Applying borders to elements (part 1 of 3).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.14: borders.html      -->
<!-- Setting borders of an element -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Borders</title>

      <style type = "text/css">

         body    { background-color: #ccffcc }

         div     { text-align: center;
                   margin-bottom: 1em;
                   padding: .5em }

         .thick  { border-width: thick }

         .medium { border-width: medium }

         .thin   { border-width: thin }

         .groove { border-style: groove }

         .inset  { border-style: inset }

         .outset { border-style: outset }

         .red    { border-color: red }

         .blue   { border-color: blue }

      </style>
   </head>

   <body>

      <div class = "thick groove">This text has a border</div>
      <div class = "medium groove">This text has a border</div>
      <div class = "thin groove">This text has a border</div>

      <p class = "thin red inset">A thin red line...</p>
      <p class = "medium blue outset">
         And a thicker blue line</p>

   </body>
</html>
```

Fig. 28.14 Applying borders to elements (part 2 of 3).

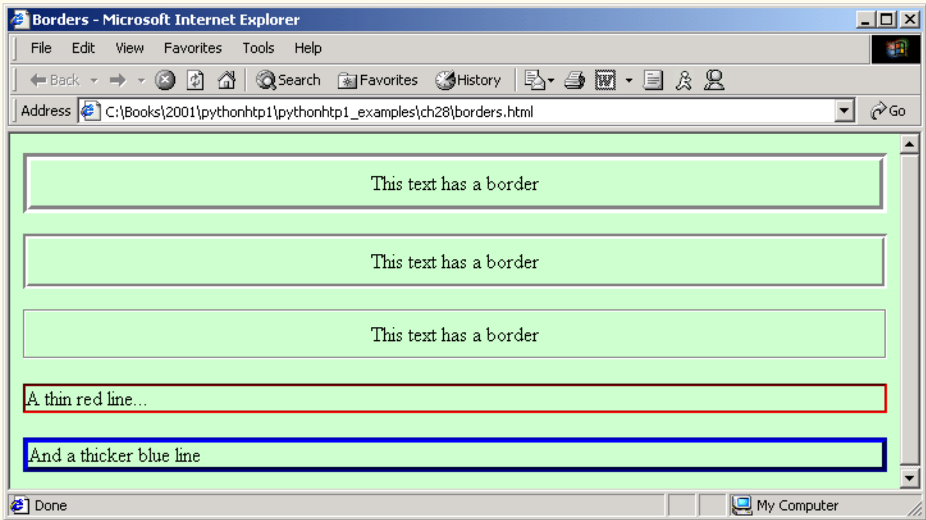


Fig. 28.14 Applying borders to elements (part 3 of 3).

In this example, we set three properties—**`border-width`**, **`border-color`** and **`border-style`**. The **`border-width`** property may be set to any of the CSS lengths or to the predefined values of ***`thin`***, ***`medium`*** or ***`thick`***. The **`border-color`** property sets the color. (This property has different meanings for different borders.)

As with **`padding`** and **`margin`**s, each of the border properties may be set for individual sides of the box (e.g., **`border-top-style`** or **`border-left-color`**). A developer can assign more than one class to an XHTML element by using the **`class`** attribute as shown in line 41.

The **`border-style`**s are ***`none`***, ***`hidden`***, ***`dotted`***, ***`dashed`***, ***`solid`***, ***`double`***, ***`groove`***, ***`ridge`***, ***`inset`*** and ***`outset`***. Borders **`groove`** and **`ridge`** have opposite effects, as do ***`inset`*** and ***`outset`***. Figure 28.15 illustrates these border styles.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.15: borders2.html  -->
<!-- Various border-styles      -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Borders</title>

      <style type = "text/css">

         body     { background-color: #ccffcc }

```

Fig. 28.15 Various **`border-style`**s (part 1 of 2).

```
         div     { text-align: center;
                   margin-bottom: .3em;
                   width: 50%;
                   position: relative;
                   left: 25%;
                   padding: .3em }
      </style>
   </head>

   <body>

      <div style = "border-style: solid">Solid border</div>
      <div style = "border-style: double">Double border</div>
      <div style = "border-style: groove">Groove border</div>
      <div style = "border-style: ridge">Ridge border</div>
      <div style = "border-style: inset">Inset border</div>
      <div style = "border-style: outset">Outset border</div>

   </body>
</html>
```

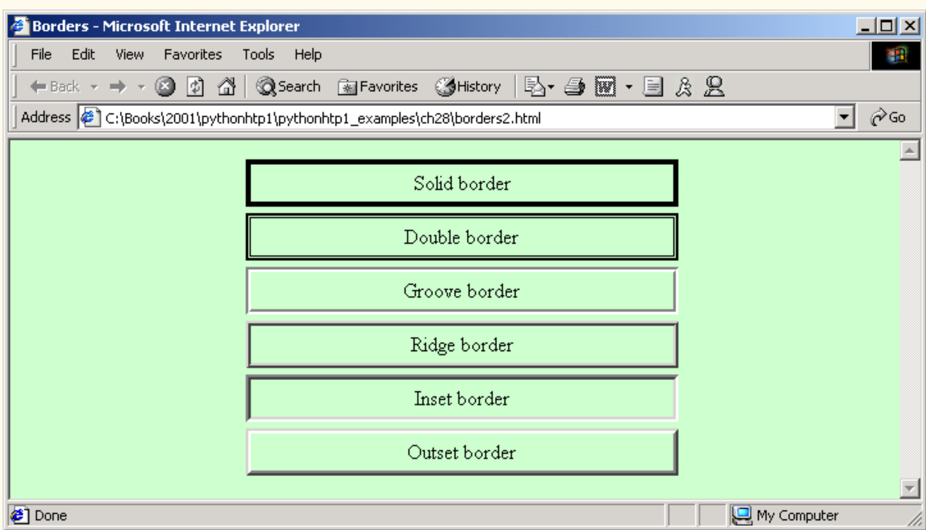


Fig. 28.15 Various **border-style**s (part 2 of 2).

28.11 User Style Sheets

Users can define their own *user style sheets* to format pages based on their preferences. For example, people with visual impairments may want to increase the page's text size. A Web-page author needs to be careful because they may inadvertently override user preferences with defined styles. This section discusses possible conflicts between *author styles* and *user styles*.

Figure 28.16 contains an author style. The **font-size** is set to **9pt** for all **<p>** tags that have class **note** applied to them.

User style sheets are external style sheets. Figure 28.17 shows a user style sheet that sets the **body**'s **font-size** to **20pt**, **color** to **yellow** and **background-color** to **#000080**.

User style sheets are not **link**ed to a document; rather, they are set in the browser's options. To add a user style sheet in Internet Explorer 5.5, select **Internet Options...**, located in the **Tools** menu. In the **Internet Options** dialog (Fig. 28.18), select **Accessibility...**, Check the **Format documents using my style sheet** check box and type the location of the user style sheet. Internet Explorer 5.5 applies the user style sheet to any document it loads.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 28.16: user_absolute.html -->
<!-- User styles                    -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>User Styles</title>

      <style type = "text/css">

         .note { font-size: 9pt }

      </style>
   </head>

   <body>

      <p>Thanks for visiting my Web site. I hope you enjoy it.
      </p><p class = "note">Please Note: This site will be
      moving soon. Please check periodically for updates.</p>

   </body>
</html>
```

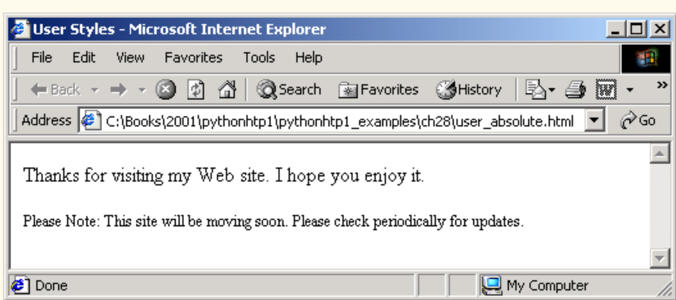


Fig. 28.16 Modifying text size with the **pt** measurement.

```
/* Fig. 28.17: userstyles.css */
/* A user stylesheet          */

body      { font-size: 20pt;
            color: yellow;
            background-color: #000080 }
```

Fig. 28.17 User style sheet.

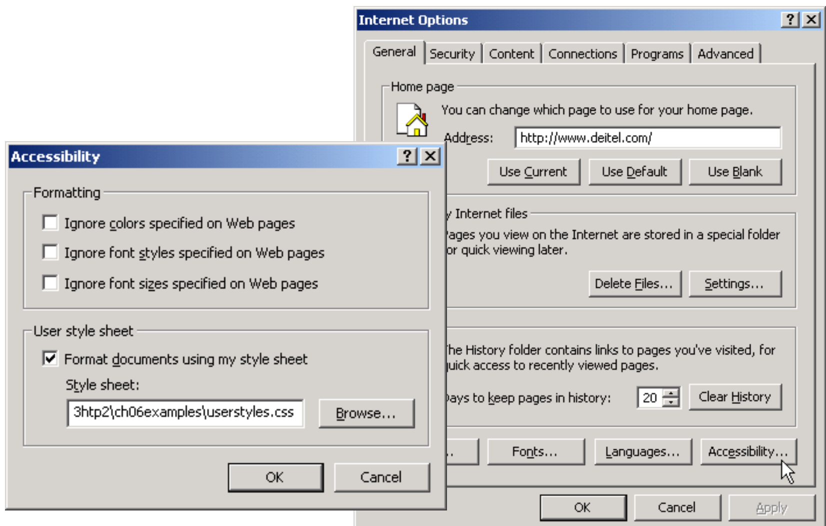


Fig. 28.18 Adding a user style sheet in Internet Explorer 5.5.

The Web page from Fig. 28.16 is displayed in Fig. 28.19, with the application of the user style sheet from Fig. 28.17.

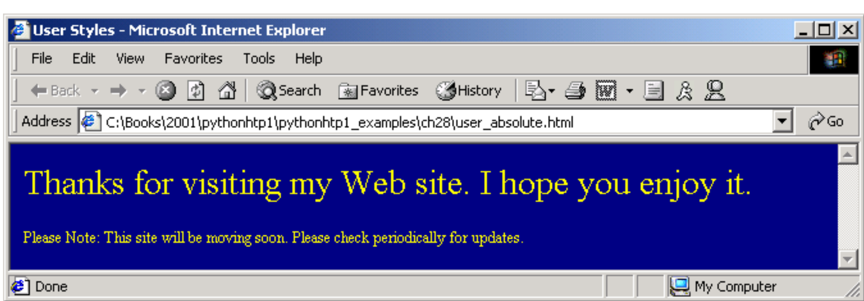


Fig. 28.19 Web page with user styles applied.

In this example if users define their own **font-size** in user style sheets, the author styles have higher precedence and override the user styles. The **9pt** font specified in the author style sheet overrides the **20pt** font specified in the user style sheet. This small font may make pages difficult to read, especially for individuals with visual impairments. A developer can avoid this problem by using relative measurements (such as **em** or **ex**) instead of absolute measurements such as **pt**. Figure 28.20 changes the **font-size** property to use a relative measurement (line 14), which does not override the user style set in Fig. 28.17. Instead, the font size displayed is relative to that specified in the user style sheet. In this case, text enclosed in the **<p>** tag displays as **20pt** and **<p>** tags that have class **note** applied to them are displayed in **15pt** (**.75** times **20pt**).

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 28.20: user_relative.html -->
6  <!-- User styles                    -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>User Styles</title>
11
12       <style type = "text/css">
13
14          .note { font-size: .75em }
15
16       </style>
17    </head>
18
19    <body>
20
21       <p>Thanks for visiting my Web site. I hope you enjoy it.
22       </p><p class = "note">Please Note: This site will be
23       moving soon. Please check periodically for updates.</p>
24
25    </body>
26 </html>
```

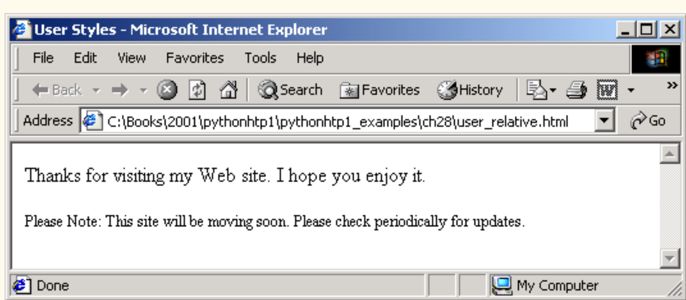


Fig. 28.20 Modifying text size with the **em** measurement.

Fig. 28.21 displays the Web page from Fig. 28.20 with the application of the user style sheet from Fig. 28.16. Notice that the second line of text displayed is larger than the same line of text in Fig. 28.19.

28.12 Internet and World Wide Web Resources

`www.w3.org/TR/REC-CSS2`
The W3C *Cascading Style Sheets, Level 2* specification contains a list of all the CSS properties. The specification also provides helpful examples detailing the use of many of the properties.

`www.webreview.com/style`
This site has several charts of CSS properties, including a list containing which browsers support what attributes and to what extent.

`tech.irt.org/articles/css.htm`
This site contains articles dealing with CSS.

`msdn.microsoft.com/workshop/author/css/site1014.asp`
This site contains samples of some CSS features.

`www.web-weaving.net`
This site contains many CSS articles.

SUMMARY

- The inline style allows a developer to declare a style for an individual element by using the **`style`** attribute in that element's opening XHTML tag.
- Each CSS property is followed by a colon and the value of the attribute.
- The **`color`** property sets text color. Color names and hexadecimal codes may be used as the value.
- Styles that are placed in the **`<style>`** tag apply to the entire document.
- **`style`** element attribute **`type`** specifies the MIME type (the specific encoding format) of the style sheet. Style sheets use **`text/css`**.
- Each rule body begins and ends with a curly brace (**`{`** and **`}`**).
- Style class declarations are preceded by a period and are applied to elements of that specific class.
- The CSS rules in a style sheet use the same format as inline styles: The property is followed by a colon (**`:`**) and the value of that property. Multiple properties are separated by semicolons (**`;`**).
- The **`background-color`** attribute specifies the background color of the element.

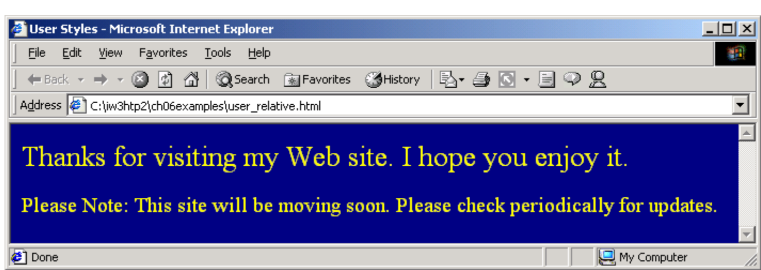


Fig. 28.21 Using relative measurements in author styles.

- The **font-family** attribute names a specific font that should be displayed. Generic font families allow authors to specify a type of font instead of a specific font, in case a browser does not support a specific font. The **font-size** property specifies the size used to render the font.
- The **class** attribute applies a style class to an element.
- Pseudoclasses provide the author access to content not specifically declared in the document. The **hover** pseudoclass is activated when the user moves the mouse cursor over an element.
- The **text-decoration** property applies decorations to text within an element, such as **underline**, **overline**, **line-through** and **blink**.
- To apply rules to multiple elements, separate the elements with commas in the style sheet.
- A pixel is a relative-length measurement: It varies in size based on screen resolution. Other relative lengths are **em**, **ex** and percentages.
- The other units of measurement available in CSS are absolute-length measurements—i.e., units that do not vary in size. These units can be **in** (inches), **cm** (centimeters), **mm** (millimeters), **pt** (points; 1 **pt**=1/72 **in**) and **pc** (picas; 1 **pc** = 12 **pt**).
- External linking can create a uniform look for a Web site; separate pages can all use the same styles. Modifying a single file makes changes to styles across an entire Web site.
- **link**'s **rel** attribute specifies a relationship between two documents.
- The CSS **position** property allows absolute positioning, which provides greater control on where elements reside. Specifying an element's **position** as **absolute** removes it from the normal flow of elements on the page and positions it according to distance from the **top**, **left**, **right** or **bottom** margins of its parent element.
- The **z-index** property allows a developer to layer overlapping elements. Elements that have higher **z-index** values are displayed in front of elements with lower **z-index** values.
- Unlike absolute positioning, relative positioning keeps elements in a general flow on the page and offsets them by the specified **top**, **left**, **right** or **bottom** values.
- Property **background-image** specifies the URL of the image, in the format **url(***fileLocation***)**. The property **background-position** places the image on the page using the values **top**, **bottom**, **center**, **left** and **right** individually or in combination for vertical and horizontal positioning. You can also position by using lengths.
- The **background-repeat** property controls the tiling of the background image. Setting the tiling to **no-repeat** displays one copy of the background image on screen. The **background-repeat** property can be set to **repeat** (the default) to tile the image vertically and horizontally, to **repeat-x** to tile the image only horizontally or to **repeat-y** to tile the image only vertically.
- The property setting **background-attachment: fixed** fixes the image in the position specified by **background-position**. Scrolling the browser window does not move the image from its set position. The default value, **scroll**, moves the image as the user scrolls the window.
- The **text-indent** property indents the first line of text in the element by the specified amount.
- The **font-weight** property specifies the "boldness" of text. Values besides **bold** and **normal** (the default) are **bolder** (bolder than **bold** text) and **lighter** (lighter than **normal** text). The value also may be justified using multiples of 100, from 100 to 900 (i.e., **100**, **200**, …, **900**). Text specified as **normal** is equivalent to **400**, and **bold** text is equivalent to **700**.
- The **font-style** property allows the developer to set text to **none**, **italic** or **oblique** (**oblique** will default to **italic** if the system does not have a separate font file for oblique text, which is normally the case).
- **span** is a generic grouping element; it does not apply any inherent formatting to its contents. Its main use is to apply styles or **id** attributes to a block of text. Element **span** is displayed inline

(an inline element) with other text and with no line breaks. A similar element is the **div** element, which also applies no inherent styles, but is displayed on a separate line, with margins above and below (a block-level element).

- The dimensions of elements on a page can be set with CSS by using the **height** and **width** properties.
- Text within an element can be **center**ed using **text-align**; other values for the **text-align** property are **left** and **right**.
- One problem with setting both dimensions of an element is that the content inside the element might sometimes exceed the set boundaries, in which case the element must be made large enough for all the content to fit. However, a developer can set the **overflow** property to **scroll**; this setting adds scroll bars if the text overflows the boundaries set for it.
- Browsers normally place text and elements on screen in the order in which they appear in the XHTML file. Elements can be removed from the normal flow of text. Floating allows you to move an element to one side of the screen; other content in the document will then flow around the floated element.
- Each block-level element has a box drawn around it, known as the box model. The properties of this box are easily adjusted.
- The **margin** property determines the distance between the element's edge and any outside text.
- CSS uses a box model to render elements on screen. The content of each element is surrounded by padding, a border and margins.
- Margins for individual sides of an element can be specified by using **margin-top**, **margin-right**, **margin-left** and **margin-bottom**.
- The padding is the distance between the content inside an element and the edge of the element. Padding can be set for each side of the box by using **padding-top**, **padding-right**, **padding-left** and **padding-bottom**.
- A developer can interrupt the flow of text around a **float**ed element by setting the **clear** property to the same direction in which the element is **float**ed—**right** or **left**. Setting the **clear** property to **all** interrupts the flow on both sides of the document.
- A property of every block-level element on screen is its border. The border lies between the padding space and the margin space and has numerous properties with which to adjust its appearance.
- The **border-width** property may be set to any of the CSS lengths or to the predefined values of **thin**, **medium** or **thick**.
- The **border-style**s available are **none**, **hidden**, **dotted**, **dashed**, **solid**, **double**, **groove**, **ridge**, **inset** and **outset**.
- The **border-color** property sets the color used for the border.
- The class attribute allows more than one class to be assigned to an XHTML element.

TERMINOLOGY

absolute positioning
absolute-length measurement
arial font
background
background-attachment
background-color
background-image
background-position

Cascading Style Sheets (CSS)
class attribute
clear property value
cm (centimeter)
colon (**:**)
color
CSS rule
cursive generic font family
dashed border-style
dotted border-style
double border-style
em (size of font)
embedded style sheet
ex (*x*-height of font)
floated element
font-style property
generic font family
groove border style
hidden border style
href attribute
in (inch)
inline style
inline-level element
inset border-style
large relative font size
larger relative font size
left
line-through text decoration
link element
linking to an external style sheet
margin
margin-bottom property
margin-left property
margin-right property
margin-top property
medium relative border width
medium relative font size
mm (millimeter)
monospace
none border-style
background-repeat
blink
block-level element
border
border-color
border-style
border-width
box model
outset border-style
overflow property

overline text decoration
padding
parent element
pc (pica)
pseudoclass
pt (point)
rel attribute (**link**)
relative positioning
relative-length measurement
repeat
ridge border-style
right
sans-serif generic font family
scroll
separation of structure from content
serif generic font family
small relative font size
smaller relative font size
solid border-style
span element
style
style attribute
style class
style in header section of the document
text flow
text/css MIME type
text-align
text-decoration property
text-indent
thick border width
thin border width
user style sheet
x-large relative font size
x-small relative font size
xx-large relative font size
xx-small relative font size
z-index

SELF-REVIEW EXERCISES

28.1 Assume that the size of the base font on a system is 12 points.
a) How big is 36-point font in **em**s?
b) How big is 8-point font in **em**s?
c) How big is 24-point font in picas?
d) How big is 12-point font in inches?
e) How big is 1-inch font in picas?

28.2 Fill in the blanks in the following statements:
a) Using the __________ element allows authors to use external style sheets in their pages.
b) To apply a CSS rule to more than one element at a time, separate the element names with a __________.
c) Pixels are a(n) __________ -length measurement unit.

d) The **hover** ________________ is activated when the user moves the mouse cursor over the specified element.
e) Setting the **overflow** property to __________ provides a mechanism for containing inner content without compromising specified box dimensions.
f) While __________ is a generic inline element that applies no inherent formatting, __________ is a generic block-level element that applies no inherent formatting.
g) Setting the **background-repeat** property to __________ tiles the specified **background-image** only vertically.
h) If you **float** an element, you can stop the flowing text by using property __________.
i) The __________ property allows you to indent the first line of text in an element.
j) Three components of the box model are the __________, __________ and __________.

ANSWERS TO SELF-REVIEW EXERCISES

28.1 a) 3 **em**s. b) 0.75 **em**s. c) 2 picas. d) 1/6 inch. e) 6 picas.

28.2 a) **link**. b) comma. c) relative. d) pseudoclass. e) **scroll**. f) **span**, **div**. g) **y-repeat**. h) **clear**. i) **text-indent**. j) padding, border, margin.

EXERCISES

28.3 Write a CSS rule that makes all text 1.5 times larger than the base font of the system and colors the text red.

28.4 Write a CSS rule that removes the underline from all links inside list items (**li**) and shifts them left by 3 **em**s.

28.5 Write a CSS rule that places a background image halfway down the page, tiling it horizontally. The image should remain in place when the user scrolls up or down.

28.6 Write a CSS rule that gives all **h1** and **h2** elements a padding of 0.5 **em**s, a **groove**d border style and a margin of 0.5 **em**s.

28.7 Write a CSS rule that changes the color of all elements containing attribute **class = "greenMove"** to green and shifts them down 25 pixels and right 15 pixels.

28.8 Write an XHTML document that shows the results of a color survey. The document should contain a form with radio buttons that allows users to vote for their favorite color. One of the colors should be selected as a default. The document should also contain a table showing various colors and the corresponding percentage of votes for each color. (Each row should be displayed in the color to which it is referring.) Use attributes to format width, border and cell spacing for the table.

28.9 Add an embedded style sheet to the XHTML document of Fig. 26.6. This style sheet should contain a rule that displays **h1** elements in blue. In addition, create a rule that displays all links in blue without underlining them. When the mouse hovers over a link, change the link's background color to yellow.

28.10 Modify the style sheet of Fig. 28.4 by changing **a:hover** to **a:hver** and **margin-left** to **margin left**. Validate the style sheet using the CSS Validator. What happens?

(***DUMP FILE***)

SELF-REVIEW EXERCISES

28.1 Assume that the size of the base font on a system is 12 points.
a) How big is 36-point font in **em**s?

ANS: 3 **em**s.
b) How big is 8-point font in **em**s?
ANS: 0.75 **em**s.
c) How big is 24-point font in picas?
ANS: 2 picas.
d) How big is 12-point font in inches?
ANS: 1/6 inch.
e) How big is 1-inch font in picas?
ANS: 6 picas.

28.2 Fill in the blanks in the following statements:
a) Using the __________ element allows you to use external style sheets in your pages.
ANS: **link**.
b) To apply a CSS rule to more than one element at a time, separate the element names with a __________.
ANS: comma.
c) Pixels are a(n) __________ -length measurement unit.
ANS: relative.
d) The **hover** ____________________ is activated when the user moves the mouse cursor over the specified element.
ANS: pseudoelement.
e) Setting the **overflow** property to __________ provides a mechanism for containing inner content without compromising specified box dimensions.
ANS: **scroll**.
f) While __________ is a generic inline element that applies no inherent formatting, __________ is a generic block-level element that applies no inherent formatting.
ANS: **span**, **div**.
g) Setting the **background-repeat** property to __________ tiles the specified **background-image** only vertically.
ANS: **y-repeat**.
h) If you **float** an element, you can stop the flowing text by using property __________.
ANS: **clear**.
i) The __________ property allows you to indent the first line of text in an element.
ANS: **text-indent**.
j) Three components of the box model are the __________, __________ and __________.
ANS: padding, border, margin.

EXERCISES

28.3 Write a CSS rule that makes all text 1.5 times larger than the base font of the system and colors it red.
ANS:

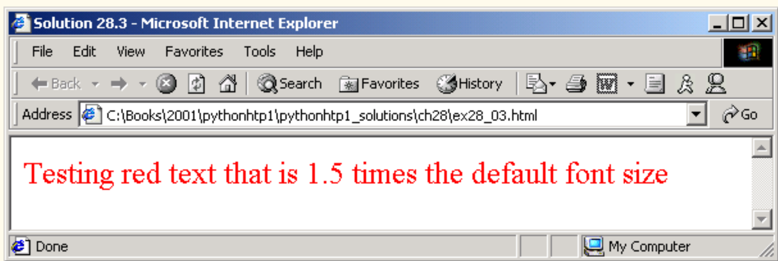
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 28.3 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
```

```
   <head>
      <title>Solution 28.3</title>
      <style type = "text/css">
         body { font-size: 1.5em;
                color: #FF0000 }
      </style>
   </head>

   <body>
      <p>Testing red text that is 1.5 times the default font
         size</p>
   </body>
</html>
```



28.4 Write a CSS rule that removes the underline from all links inside list items (**li**) and shifts them left by 3 **em**s.

ANS:

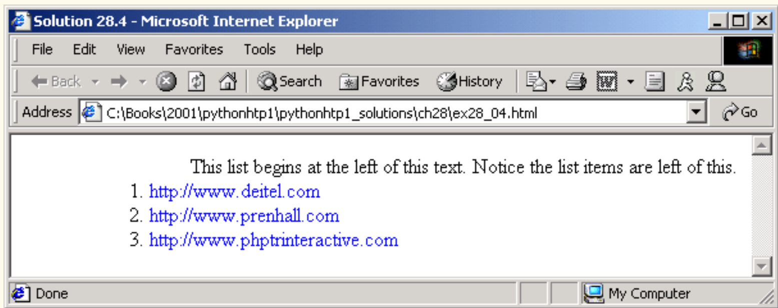
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 28.4 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 28.4</title>
         <style type = "text/css">
            li a { text-decoration: none }
            li   { position: relative;
                   left: -3em }
            ol   { position: absolute;
                   left: 100px }
         </style>
   </head>

   <body>
      <ol>
         This list begins at the left of this text.
         Notice the list items are left of this.
         <li><a href = "http://www.deitel.com">
```

```
                  http://www.deitel.com</a></li>
               <li><a href = "http://www.prenhall.com">
                  http://www.prenhall.com</a></li>
               <li><a href = "http://www.phptrinteractive.com">
                  http://www.phptrinteractive.com</a></li>
            </ol>
      </body>
   </html>
```



28.5 Write a CSS rule that places a background image halfway down the page, tiling it horizontally. The image should remain in place when the user scrolls up or down.

ANS:

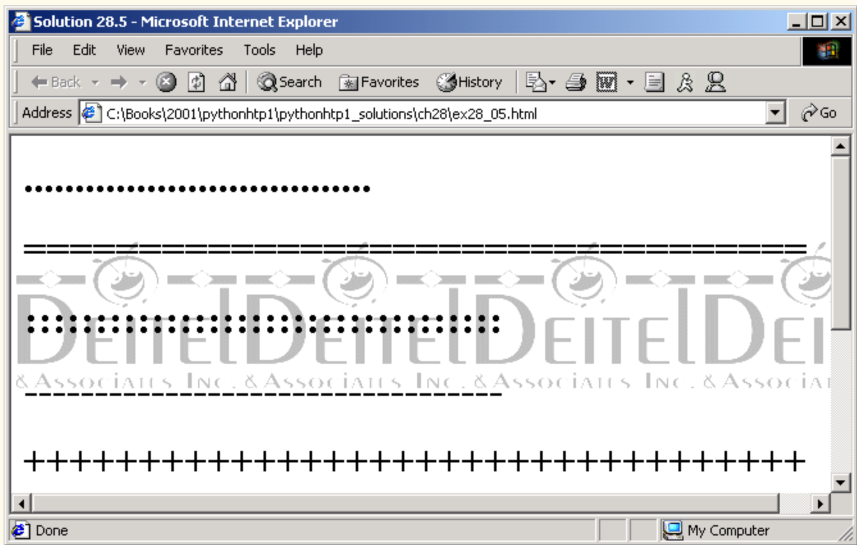
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

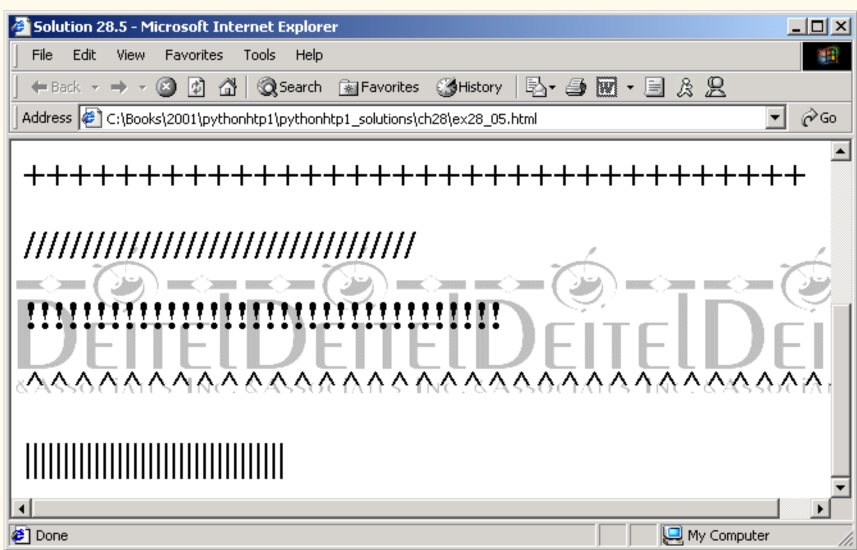
<!-- Exercise 28.5 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 28.5</title>
      <style type = "text/css">
         body { background-image: url(logo.gif);
                background-position: left center;
                background-repeat: repeat-x;
                background-attachment: fixed }
      </style>
   </head>

   <body>
      <h1>....................................</h1>
      <h1>====================================</h1>
      <h1>::::::::::::::::::::::::::::::::::::</h1>
      <h1>------------------------------------</h1>
      <h1>++++++++++++++++++++++++++++++++++++</h1>
      <h1>////////////////////////////////////</h1>
      <h1>!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!</h1>
      <h1>^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^</h1>
```

```
        <h1>||||||||||||||||||||||||||||||||||</h1>
    </body>
</html>
```





28.6 Write a CSS rule that gives all **`h1`** and **`h2`** elements a padding of 0.5 **`em`**s, a **`groove`**d border style and a margin of 0.5 **`em`**s.

ANS:

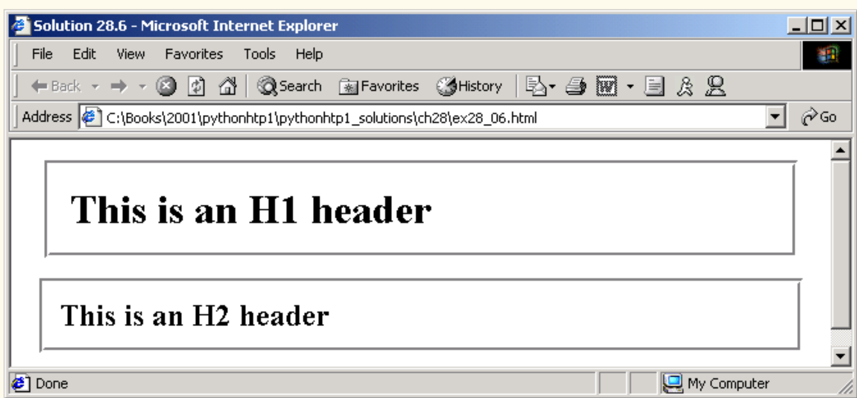
```
<?xml version = "1.0"?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 6.6 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 6.6</title>
      <style type = "text/css">
         h1, h2 { padding: .5em;
                  border-style: groove;
                  margin: .5em }
      </style>
   </head>

   <body>
      <h1>This is an H1 header</h1>
      <h2>This is an H2 header</h2>
   </body>
</html>
```



28.7 Write a CSS rule that changes the color of all elements with attribute **class = "greenMove"** to green and shifts them down 25 pixels and right 15 pixels.

ANS:

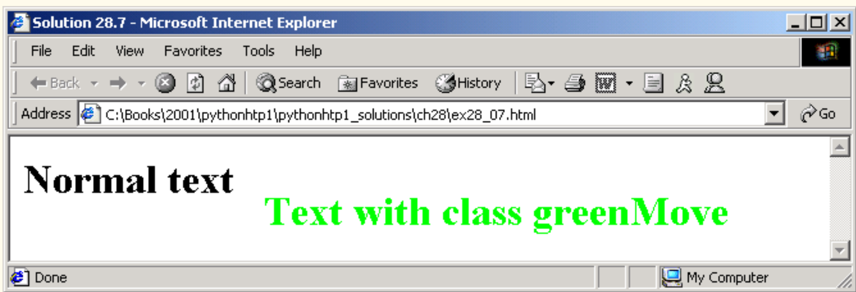
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 28.7 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 28.7</title>
      <style type = "text/css">
         .greenMove { color: #00FF00;
```

```
                             position: relative;
                             top: 25px;
                             left: 15px }
      </style>
   </head>

   <body>
      <h1>Normal text
         <span class = "greenMove">Text with class
            greenMove
         </span>
      </h1>
   </body>
</html>
```



28.8 Write an XHTML document showing the results of a survey of people's favorite color. The document should contain a form with radio buttons that allows users to vote for their favorite color. One of the colors should be selected as a default. The document should also contain a table showing various colors and the corresponding percentage of votes for each color. (Each row should be displayed in the color to which it is referring.) Use attributes to format width, border and cell spacing for the table. Validate the document against an appropriate XHTML DTD.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

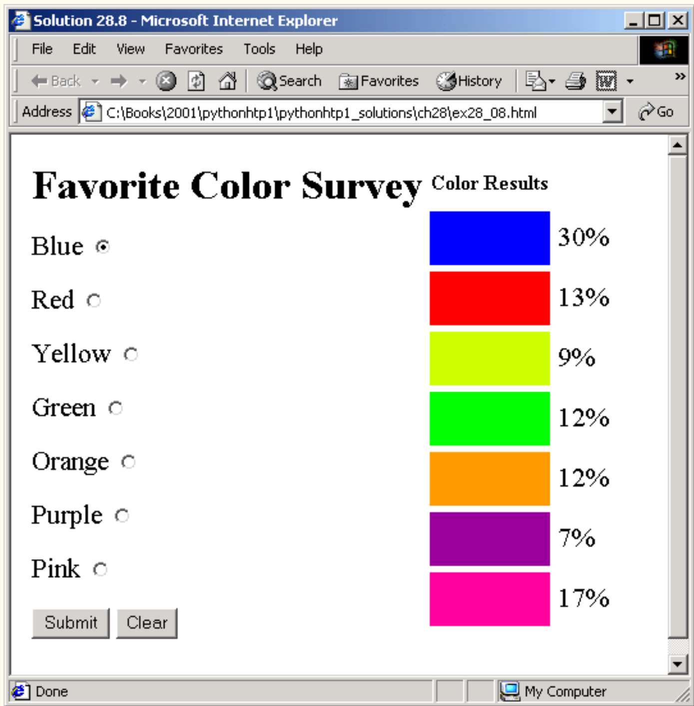
<!-- Exercise 28.8 Solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 28.8</title>
      <style type = "text/css">
         .blue   { background-color: #0000ff }
         .red    { background-color: #ff0000 }
         .yellow { background-color: #ccff00 }
         .green  { background-color: #00ff00 }
         .orange { background-color: #ff9900 }
         .purple { background-color: #990099 }
```

```
         .pink   { background-color: #ff0099 }
         p       { font-size: 16pt; text-decoration: bold }
      </style>
   </head>
   <body>

      <table border = "0" cellspacing = "5">
         <tr>
            <td rowspan = "9">
               <h1>Favorite Color Survey</h1>
               <form method = "post" action = "">
                  <p>
                     <label>Blue
                        <input name = "color" type = "radio"
                           checked = "checked" value = "Blue" />
                     </label>
                  </p>
                  <p>
                     <label>Red
                     <input name = "color" type = "radio"
                        value = "Red" />
                     </label>
                  </p>
                  <p><label>Yellow
                     <input name = "color" type = "radio"
                        value = "Yellow" />
                     </label>
                  </p>
                  <p>
                     <label>Green
                     <input name = "color" type = "radio"
                        value = "Green" />
                     </label>
                  </p>
                  <p>
                     <label>Orange
                     <input name = "color" type = "radio"
                        value = "Orange" />
                     </label>
                  </p>
                  <p>
                     <label>Purple
                     <input name = "color" type = "radio"
                        value = "Purple" />
                     </label>
                  </p>
                  <p>
                     <label>Pink
                     <input name = "color" type = "radio"
                        value = "Pink" />
                     </label>
                  </p>
                  <p>
                     <input type = "submit" value =
```

```
                          "Submit" />
                     <input type = "reset" value = "Clear" />
                  </p>
               </form>
            </td>
            <td><strong>Color Results</strong></td>
         </tr>
         <tr>
            <td class = "blue" ></td>
            <td><p>30%</p></td>
         </tr>
         <tr>
            <td class = "red">
            </td>
            <td><p>13%</p></td>
         </tr>
         <tr>
            <td class = "yellow"></td>
            <td><p>9%</p></td>
         </tr>
         <tr>
            <td class = "green"></td>
            <td><p>12%</p></td>
         </tr>
         <tr>
            <td class = "orange"></td>
            <td><p>12%</p></td>
         </tr>
         <tr>
            <td class = "purple"></td>
            <td><p>7%</p></td>
         </tr>
         <tr>
            <td class = "pink"></td>
            <td><p>17%</p></td>
         </tr>
         <tr>
            <td></td>
         </tr>
      </table>
   </body>
</html>
```



28.9 Add an embedded style sheet to the XHTML document of Fig. 26.4. This style sheet should contain a rule that displays **h1** elements in blue. In addition, create a rule that displays all links in blue without underlining them. When the mouse hovers over a link, change the link's background color to yellow.

ANS:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 28.9: solution -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Solution 28.9</title>
      <style type = "text/css">
         h1       { color: blue }
         .dark    { font-weight: bold }
         a        { text-decoration: none }
         a:hover { background-color: #FFFF00 }
      </style>
   </head>

   <body>

```

```
         <h1>Here are my favorite sites</h1>

         <p><span class = "dark">Click on a name to go to
            that page.</span></p>

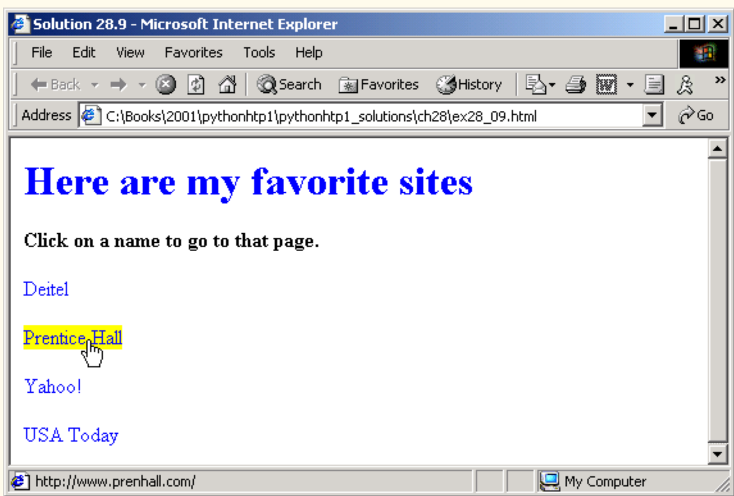
         <p><a href = "http://www.deitel.com">Deitel</a></p>

         <p><a href = "http://www.prenhall.com">Prentice
            Hall</a></p>

         <p><a href = "http://www.yahoo.com">Yahoo!</a></p>

         <p><a href = "http://www.usatoday.com">USA Today</a></p>

      </body>
   </html>
```



28.10 Modify the style sheet of Fig. 28.4 by changing **`a:hover`** to **`a:hver`** and **`margin-left`** to **`margin left`**. Validate the style sheet using the CSS Validator. What happens?

ANS:

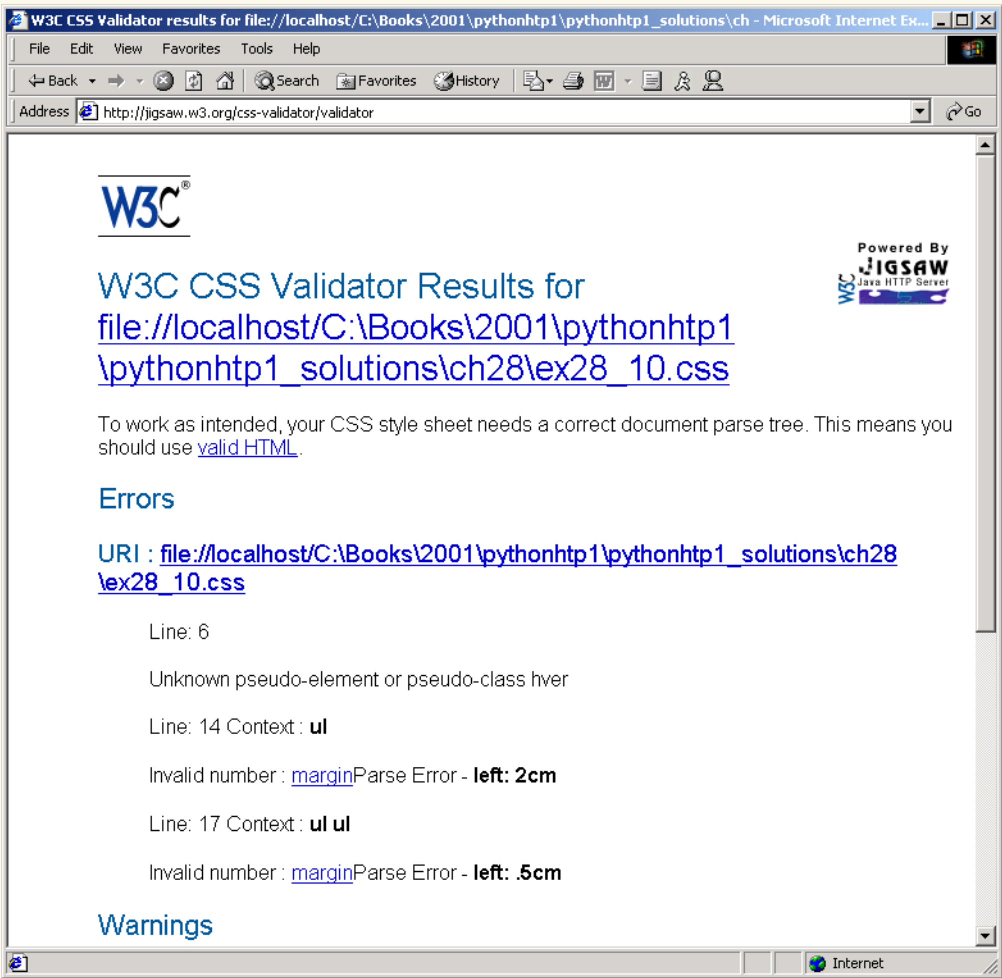
```
/* Exercise 28.10: modified Fig 28.4 styles.css */
/* An external stylesheet */

a        { text-decoration: none }

a:hver   { text-decoration: underline;
           color: red;
           background-color: #ccffcc }

li em    { color: red;
           font-weight: bold;
```

```
12                background-color: #ffffff }
13
14  ul        { margin left: 2cm }
15
16  ul ul     { text-decoration: underline;
17              margin left: .5cm }
```



[***Notes To Reviewers***]

- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send us e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **cheryl.yaeger@deitel.net**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
- Please be constructive. This book will be published soon. We all want to publish the best possible book.
- If you find something that is incorrect, please show us how to correct it.
- Please read all the back matter including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

S

T

U

V

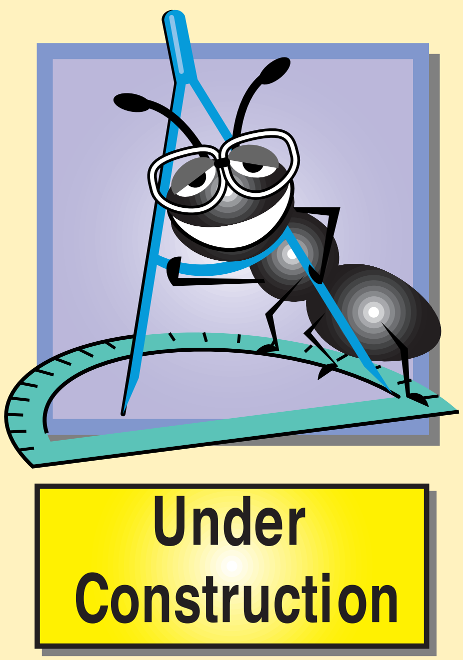
W

X

Z

29

PHP



Objectives

- To understand PHP data types, operators, arrays and control structures.
- To understand string processing and regular expressions in PHP.
- To construct programs that process form data.
- To read and write client data using cookies.
- To construct programs that interact with MySQL databases.

Conversion for me was not a Damascus Road experience. I slowly moved into an intellectual acceptance of what my intuition had always known.
Madeleine L'Engle

Be careful when reading health books; you may die of a misprint.
Mark Twain

Reckeners without their host must recken twice.
John Heywood

There was a door to which I found no key; There was the veil through which I might not see.
Omar Khayyam

[***Notes to Reviewers***]

- Our first topic choice for this chapter was PHP. We then decided to change the topic to PSP (since it is more Python specific); however, due to the fact that PSP is in poor condition, we decided to revert back to PHP. We still prefer to do PSP, so any information you can provide would be helpful. Websites? Documentation? We were unable to get PSP 1.3 to run with Tomcat or JRun. Then, we were unable to download PSP 1.3.
- Example 29.19 is not working (it is unable to open the Products database). We are currently working to resolve this issue.
- **Please mark your comments in place on a paper copy of the chapter.**
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **`rashmi.jayaprakash@deitel.net`** and **`ben.wiedermann@deitel.net`**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copy edited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are mostly concerned with technical correctness and correct use of idiom. We will not make significant adjustments to our writing or coding style on a global scale. Please send us a short e-mail if you would like to make a suggestion.
- If you find something incorrect, please show us how to correct it.
- In the later round(s) of review, please read all the back matter, including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

Outline

29.1 Introduction

PHP, or *PHP Hypertext Preprocessor*, is quickly becoming one of the most popular server-side scripting languages for creating dynamic Web pages. PHP was created in 1994 by Rasmus Lerdorf (who currently works for Linuxcare Inc. as a Senior Open-Source Researcher) to track users at his Web site.[1] In 1995, Lerdorf released it as a package called the "Personal Home Page Tools." PHP 2 featured built-in database support and form handling. In 1997, PHP 3 was released, featuring a rewritten parser, which substantially increased performance and led to an explosion in PHP use. It is estimated that over six million domains now use PHP. The release of PHP 4, which features the new *Zend Engine* and is much faster and more powerful than its predecessor, should further increase PHP's popularity.[2] More information about the Zend engine can be found at **`www.zend.com`**.

PHP is an *open-source* technology that is supported by a large community of users and developers. Open source software provides developers with access to the software's source code and free redistribution rights. PHP is platform independent; implementations exist for all major UNIX, Linux and Windows operating systems. PHP also provides support for a large number of databases, including MySQL.

After introducing the basics of the scripting language, we discuss viewing environment variables. Knowing information about a client's execution environment allows dynamic content to be sent to the client. We then discuss form processing and business logic, which are vital to e-commerce applications. We provide an example of implementing a private Web site through username and password verification. Next, we build a three-tier, Web-based application that queries a MySQL database. Finally, we show how Web sites use cookies to store information on the client that will be retrieved during a client's subsequent visits to a Web site.

29.2 PHP

When the World Wide Web and Web browsers were introduced, the Internet began to achieve widespread popularity. This greatly increased the volume of requests for information from Web servers. The power of the Web resides not only in serving content to users, but also in responding to requests from users and generating Web pages with dynamic content. It became evident that the degree of interactivity between the user and the server would be crucial. While other languages can perform this function as well, PHP was written specifically for interacting with the Web.

PHP code is embedded directly into XHTML documents. This allows the document author to write XHTML in a clear, concise manner, without having to use multiple **print** statements, as is necessary with other CGI-based languages. Figure 29.1 presents a simple PHP program that displays a welcome message.

In PHP, code is inserted between the scripting delimiters ***<?php*** and **?>**. PHP code can be placed anywhere in XHTML markup, as long as the code is enclosed in these scripting delimiters. Line 8 declares variable **$name** and assigns to it the string **"Paul"**. All variables are preceded by the ***$*** *special symbol* and are created the first time they are encountered by the PHP interpreter. PHP statements are terminated with a *semicolon* (***;***).

Common Programming Error 29.1

*Failing to precede a variable name with a **$** is a syntax error.*

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <!-- Fig. 29.1: fig29_01.php -->
5  <!-- Our first PHP script    -->
6
7  <?php
8     $name = "Paul";    // declaration
9  ?>
10
11 <html xmlns = "http://www.w3.org/1999/xhtml">
12    <head>
13       <title>A simple PHP document</title>
14    </head>
15
16    <body style = "font-size: 2em">
17       <p>
18          <strong>
19
20             <!-- print variable name's value -->
21             Welcome to PHP, <?php print( "$name" ); ?>!
22          </strong>
23       </p>
24    </body>
25 </html>
```

Fig. 29.1 Simple PHP program (part 1 of 2).

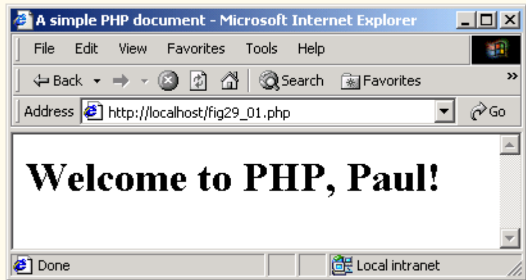


Fig. 29.1 Simple PHP program (part 2 of 2).

Common Programming Error 29.2

Variable names in PHP are case sensitive. Failure to use the proper mixture of case is a syntax error.

Common Programming Error 29.3

Forgetting to terminate a statement with a semicolon (`;`) is a syntax error.

Line 8 contains a *single-line comment*, which begins with two *forward slashes* (`//`). Text to the right of the slashes is ignored by the interpreter. Comments can also begin with the pound sign (`#`). Multiline comments begin with delimiter `/*` and end with delimiter `*/`.

Line 21 outputs the value of variable **`$name`** by calling function **`print`**. The actual value of **`$name`** is printed, instead of **`"$name"`**. When a variable is encountered inside a double-quoted (`""`) string, PHP *interpolates* the variable. In other words, PHP inserts the variable's value where the variable name appears in the string. Thus, variable **`$name`** is replaced by **`Paul`** for printing purposes. PHP variables are "*multitype*", meaning that they can contain different types of data (e.g., *integer*s, *doubles* or *strings*) at different times. Figure 29.2 introduces these data types.

Data type	Description
Integer	Whole numbers (i.e., numbers without a decimal point).
Double	Real numbers (i.e., numbers containing a decimal point).
String	Text enclosed in either single (`''`) or double (`""`) quotes.
Boolean	True or false.
Array	Group of elements of the same type.
Object	Group of associated data and methods.
Resource	An external data source.
Null	No value.

Fig. 29.2 PHP data types.

Good Programming Practice 29.1

Whitespace enhances the readability of PHP code. It also simplifies programming and debugging.

PHP scripts usually end with **.php**, although a server can be configured to handle other file extensions. To run a PHP script, PHP must first be installed on your system. Visit **www.deitel.com** for PHP installation and configuration instructions. Although PHP can be used from the command line, a Web server is necessary to take full advantage of the scripting language. Figure 29.3 demonstrates the PHP data types introduced in Fig. 29.2.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.3: fig29_03.php         -->
<!-- Demonstration of PHP data types -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>PHP data types</title>
   </head>

   <body>

      <?php

         // declare a string, double and integer
         $testString = "3.5 seconds";
         $testDouble = 79.2;
         $testInteger = 12;
      ?>

      <!-- print each variable's value -->
      <?php print( $testString ) ?> is a string.<br />
      <?php print( $testDouble ) ?> is a double.<br />
      <?php print( $testInteger ) ?> is an integer.<br />

      <br />
      Now, converting to other types:<br />
      <?php

         // call function settype to convert variable
         // testString to different data types
         print( "$testString" );
         settype( $testString, "double" );
         print( " as a double is $testString <br />" );
         print( "$testString" );
         settype( $testString, "integer" );
         print( " as an integer is $testString <br />" );
         settype( $testString, "string" );
         print( "Converting back to a string results in
            $testString <br /><br />" );
```

Fig. 29.3 Type conversion example (part 1 of 2).

```
42
43          $value = "98.6 degrees";
44
45          // use type casting to cast variables to a
46          // different type
47          print( "Now using type casting instead: <br />
48             As a string - " . (string) $value .
49             "<br />As a double - " . (double) $value .
50             "<br />As an integer - " . (integer) $value );
51       ?>
52    </body>
53 </html>
```

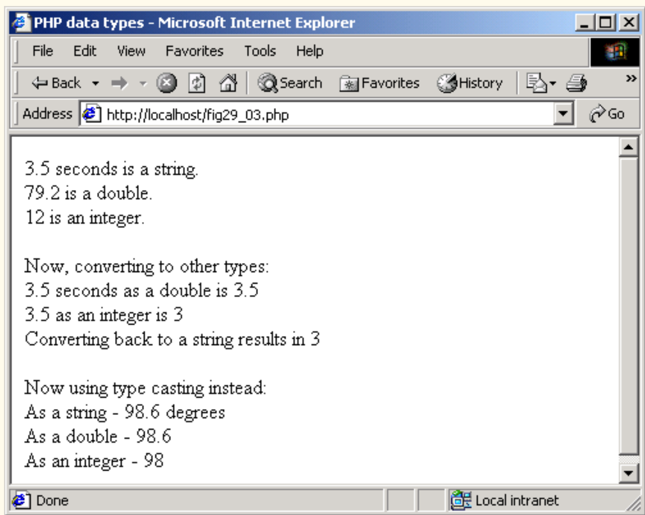


Fig. 29.3 Type conversion example (part 2 of 2).

Conversion between different data types may be necessary when performing arithmetic operations with variables. In PHP, data-type conversion can be performed by passing the data type as an argument to *function* ***settype***. Lines 17–19 assign a string to variable **$testString**, a double to variable **$testDouble** and an integer to variable **$testInteger**. Variables are converted the to data type of the value they are assigned. For example, variable **$testString** becomes a string when assigned the value **"3.5 seconds"**. Lines 23–25 **print** the value of each variable. Notice that the enclosing of a variable name in double quotes in a **print** statement is optional. Lines 34–39 call function **settype** to modify the data type of each variable. Function **settype** takes two arguments: The variable whose data type is to be changed and the variable's new data type. Calling function **settype** can result in loss of data. For example, doubles are truncated when they are converted to integers. When converting between a string and a number, PHP uses the value of the number that appears at the beginning of the string. If no number appears at the beginning of the string, the string evaluates to **0**. In line 34, the string **"3.5 seconds"** is converted to a double, resulting in the value **3.5** being stored in variable

$testString. In line 37, double **3.5** is converted to integer **3**. When we convert this variable to a string (line 39), the variable's value becomes **"3"**.

Another option for conversion between types is *casting* (or *type casting*). Unlike **settype**, casting does not change a variable's content. Rather, type casting creates a temporary copy of a variable's value in memory. Lines 47–50 cast variable **$data**'s value to a **string**, a **double** and an **integer**. Type casting is necessary when a specific data type is required for an arithmetic operation.

The *concatenation operator* (**.**) concatenates strings. This combines multiple strings in the same **print** statement (lines 47–50). A **print** statement may be split over multiple lines; everything that is enclosed in the parentheses, terminated by a semicolon, is sent to the client. PHP provides a variety of arithmetic operators, which we demonstrate in Fig. 29.4.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.4: fig29_04.php    -->
<!-- Demonstration of operators -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Using arithmetic operators</title>
   </head>

   <body>
      <?php
         $a = 5;
         print( "The value of variable a is $a <br />" );

         // define constant VALUE
         define( "VALUE", 5 );

         // add constant VALUE to variable $a
         $a = $a + VALUE;
         print( "Variable a after adding constant VALUE
            is $a <br />" );

         // multiply variable $a by 2
         $a *= 2;
         print( "Multiplying variable a by 2 yields $a <br />" );

         // test if variable $a is less than 50
         if ( $a < 50 )
            print( "Variable a is less than 50 <br />" );

         // add 40 to variable #a
         $a += 40;
         print( "Variable a after adding 40 is $a <br />" );

         // test if variable $a is 50 or less
         if ( $a < 51 )
            print( "Variable a is still 50 or less<br />" );
```

Fig. 29.4 Using PHP's arithmetic operators (part 1 of 2).

```
40
41          // test if variable $a is between 50 and 100, inclusive
42          elseif ( $a < 101 )
43             print( "Variable a is now between 50 and 100,
44                inclusive<br />" );
45          else
46             print( "Variable a is now greater than 100
47                <br />" );
48
49          // add 10 to constant VALUE
50          $test = 10 + VALUE;
51          print( "A constant plus constant
52             VALUE yields $test <br />" );
53
54          // add a string to an integer
55          $str = "3 dollars";
56          $a += $str;
57          print( "Adding a string to an integer yields $a
58             <br />" );
59       ?>
60    </body>
61 </html>
```

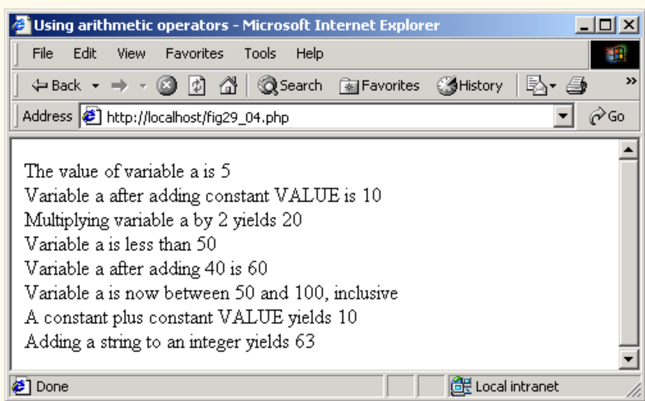


Fig. 29.4 Using PHP's arithmetic operators (part 2 of 2).

Line 14 declares variable **`$a`** and assigns it the value **`5`**. Line 18 calls function **`define`** to create a *named constant*. A constant is a value that cannot be modified once it is declared. Function **`define`** takes two arguments: the name and value of the constant. An optional third argument accepts a boolean value that specifies whether the constant is case insensitive—constants are case sensitive by default.

Common Programming Error 29.4

Assigning a value to a constant after a constant is declared is a syntax error.

Line 21 adds constant **`VALUE`** to variable **`$a`**, which is a typical use of arithmetic operators. Line 26 uses the *assignment operator* **`*=`** to yield an expression equivalent to **`$a =`**

$a * 2 (thus assigning **$a** the value **20**). These assignment operators (i.e., **+=**, **-=**, ***=** and **/=**) are syntactical shortcuts. Line 34 adds **40** to the value of variable **$a**.

Testing and Debugging Tip 29.1

Always initialize variables before using them. Doing so helps avoid subtle errors.

Strings are converted to integers when they are used in arithmetic operations (lines 54–55). In line 55, the string value **"3 dollars"** is converted to the integer **3** before being added to integer variable **$a**.

Testing and Debugging Tip 29.2

*Function **print** can be used to display the value of a variable at a particular point during a program's execution. This is often helpful in debugging a script.*

Common Programming Error 29.5

*Using an uninitialized variable might result in an incorrect numerical calculation. For example, multiplying a number by an uninitialized variable results in **0**.*

The words **if**, **elseif** and **else** are PHP *keywords* (Fig. 29.5), meaning that they are reserved for implementing language features. PHP provides the capability to store data in arrays. Arrays are divided into *elements* that behave as individual variables. Figure 29.6 demonstrates techniques for array initialization and manipulation.

PHP keywords					
and	do	for	include	require	true
break	else	foreach	list	return	var
case	elseif	function	new	static	virtual
class	extends	global	not	switch	xor
continue	false	if	or	this	while
default					

Fig. 29.5 PHP keywords.

Individual array elements are accessed by following the array-variable name with an index enclosed in braces (**[]**). If a value is assigned to an array that does not exist, then the array is created (line 18). Likewise, assigning a value to an element where the index is omitted appends a new element to the end of the array (line 21). The **for** loop (lines 24–25) **print**s each element's value. *Function **count*** returns the total number of elements in the array. Because array indices start at 0, the index of the last element is one less than the total number of elements. In this example, the **for** loop terminates once the counter (**$i**) is equal to the number of elements in the array.

Line 31 demonstrates a second method of initializing arrays. Function ***array*** returns an array that contains the arguments passed to it. The first item in the list is stored as the first array element, the second item is stored as the second array element, and so on. Lines 32–33 use another **for** loop to print out each array element's value.

In addition to integer indices, arrays can have nonnumeric indices (lines 39–41). For example, indices **Harvey**, **Paul** and **Tem** are assigned the values **21**, **18** and **23**, respec-

tively. PHP provides functions for *iterating* through the elements of an array (lines 45–46). Each array has a built-in *internal pointer*, which points to the array element currently being referenced. *Function* ***reset*** sets the iterator to the first element of the array. Function ***key*** returns the index of the element to which the iterator points, and function ***next*** moves the iterator to the next element. The **for** loop continues to execute as long as function **key** returns an index. Function **next** returns **false** when there are no additional elements in the array. When this occurs, function **key** cannot return an index, and the script terminates. Line 47 **print**s the index and value of each element.

Function **array** can also be used to initialize arrays with string indices. In order to override the automatic numeric indexing performed by function **array**, use operator **=>** as demonstrated on lines 54–61. The value to the left of the operator is the array index, and the value to the right is the element's value.

The ***foreach*** *loop* is a control structure that is specially designed for iterating through arrays (line 64). The syntax for a **foreach** loop starts with the array to iterate through, followed by the keyword ***as***, followed by the variables to receive the index and the value for each element. We use the **foreach** loop to **print** each element and value of array **$fourth**.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <!-- Fig. 29.6: fig29_06.php -->
5  <!-- Array manipulation      -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Array manipulation</title>
10    </head>
11
12    <body>
13       <?php
14
15          // create array first
16          print( "<strong>Creating the first array</strong>
17             <br />" );
18          $first[ 0 ] = "zero";
19          $first[ 1 ] = "one";
20          $first[ 2 ] = "two";
21          $first[] = "three";
22
23          // print each element's index and value
24          for ( $i = 0; $i < count( $first ); $i++ )
25             print( "Element $i is $first[$i] <br />" );
26
27          print( "<br /><strong>Creating the second array
28             </strong><br />" );
29
30          // call function array to create array second
31          $second = array( "zero", "one", "two", "three" );
32          for ( $i = 0; $i < count( $second ); $i++ )
```

Fig. 29.6 Array manipulation (part 1 of 3).

```
               print( "Element $i is $second[$i] <br />" );

            print( "<br /><strong>Creating the third array
               </strong><br />" );

            // assign values to non-numerical indices
            $third[ "Harvey" ] = 21;
            $third[ "Paul" ] = 18;
            $third[ "Tem" ] = 23;

            // iterate through the array elements and print each
            // element's name and value
            for ( reset( $third ); $element = key( $third );
               next( $third ) )
               print( "$element is $third[$element] <br />" );

            print( "<br /><strong>Creating the fourth array
               </strong><br />" );

            // call function array to create array fourth using
            // string indices
            $fourth = array(
               "January"   => "first",   "February" => "second",
               "March"     => "third",   "April"    => "fourth",
               "May"       => "fifth",   "June"     => "sixth",
               "July"      => "seventh", "August"   => "eighth",
               "September" => "ninth",   "October"  => "tenth",
               "November"  => "eleventh","December" => "twelfth"
               );

            // print each element's name and value
            foreach ( $fourth as $element => $value )
               print( "$element is the $value month <br />" );
         ?>
      </body>
   </html>
```

Fig. 29.6 Array manipulation (part 2 of 3).

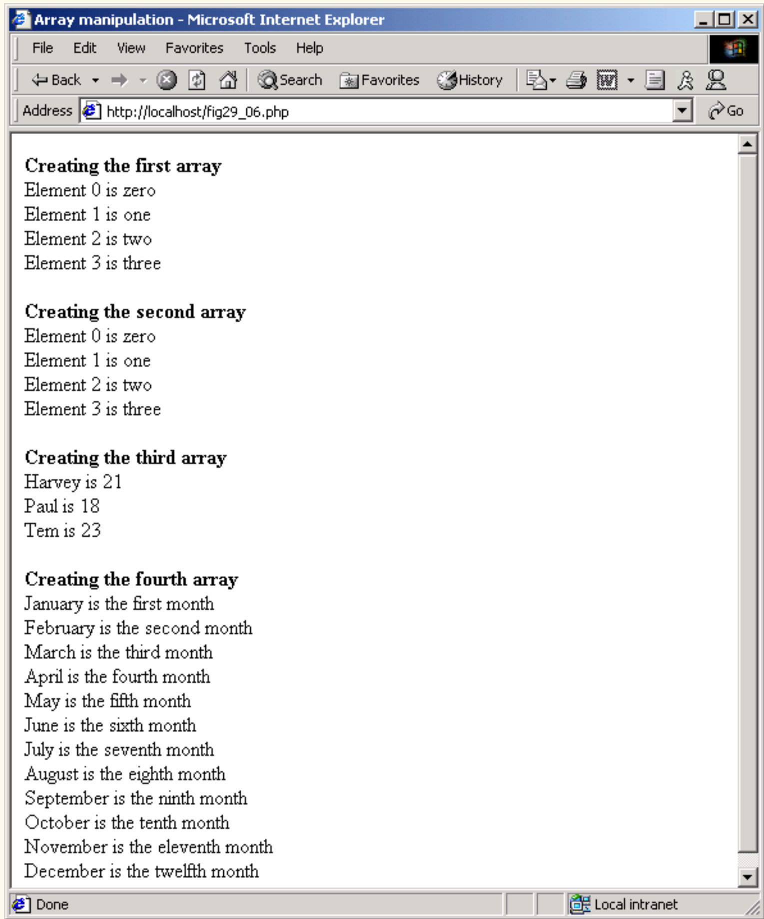


Fig. 29.6 Array manipulation (part 3 of 3).

29.3 String Processing and Regular Expressions

PHP processes text data easily and efficiently, enabling straightforward searching, substitution, extraction and concatenation of strings. Text manipulation in PHP is usually done with *regular expressions*—a series of characters that serve as pattern-matching templates (or search criteria) in strings, text files and databases. This feature allows complex searching and string processing to be performed using relatively simple expressions.

Many string-processing tasks are accomplished by using PHP's *equality* and *comparison* operators (Fig. 29.7). Line 16 declares and initializes array **`$fruits`** by calling function **`array`**. Lines 19–40 iterate through the array, comparing the array's elements to one another.

Lines 23 and 25 call function ***`strcmp`*** to compare two strings. If the first string alphabetically precedes the second string, then **`-1`** is returned. If the strings are equal, then **`0`** is returned. If the first string alphabetically follows the second string, then **`1`** is returned. The **`for`** loop (line 19) iterates through each element in the **`$fruits`** array. Lines 23–29 com-

pare each element to the string **"banana"**, printing the elements that are greater than, less than and equal to the string.

Relational operators (**==**, **!=**, **<**, **<=**, **>** and **>=**) can also be used to compare strings. Lines 33–38 use relational operators to compare each element of the array to the string **"apple"**. These operators are also used for numerical comparison with integers and doubles.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <!-- Fig. 29.7: fig29_07.php -->
5  <!-- String Comparison       -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>String Comparison</title>
10    </head>
11
12    <body>
13       <?php
14
15          // create array fruits
16          $fruits = array( "apple", "orange", "banana" );
17
18          // iterate through each array element
19          for ( $i = 0; $i < count( $fruits ); $i++ ) {
20
21             // call function strcmp to compare the array element
22             // to string "banana"
23             if ( strcmp( $fruits[ $i ], "banana" ) < 0 )
24                print( $fruits[ $i ]." is less than banana " );
25             elseif ( strcmp( $fruits[ $i ], "banana" ) > 0 )
26                print( $fruits[ $i ].
27                   " is greater than banana " );
28             else
29                print( $fruits[ $i ]." is equal to banana " );
30
31             // use relational operators to compare each element
32             // to string "apple"
33             if ( $fruits[ $i ] < "apple" )
34               print( "and less than apple! <br />" );
35             elseif ( $fruits[ $i ] > "apple" )
36               print( "and greater than apple! <br />" );
37             elseif ( $fruits[ $i ] == "apple" )
38                print( "and equal to apple! <br />" );
39
40          }
41       ?>
42    </body>
43 </html>
```

Fig. 29.7 Using the string comparison operators (part 1 of 2).

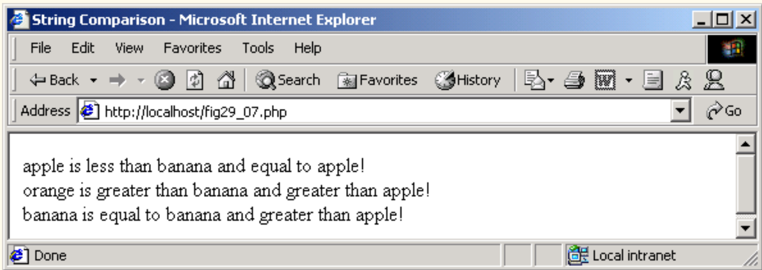


Fig. 29.7 Using the string comparison operators (part 2 of 2).

For more powerful string comparisons, PHP provides functions **`ereg`** and **`preg_match`**, which use regular expressions to search a string for a specified pattern. Function **`ereg`** uses *Portable Operating System Interface (POSIX) extended regular expressions*, whereas function **`preg_match`** provides *Perl-compatible regular expressions*. POSIX-extended regular expressions are a standard to which PHP regular expressions conform. In this section, we use function **`ereg`**. Perl regular expressions are more widely used than POSIX regular expressions. Support for Perl regular expressions also eases migration from Perl to PHP. Consult PHP's documentation for a list of differences between the Perl and PHP implementations. Figure 29.8 demonstrates some of PHP's regular expression capabilities.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.8: fig29_08.php   -->
<!-- Using regular expressions -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Regular expressions</title>
   </head>

   <body>
      <?php
         $search = "Now is the time";
         print( "Test string is: '$search'<br /><br />" );

         // call function ereg to search for pattern 'Now'
         // in variable search
         if ( ereg( "Now", $search ) )
            print( "String 'Now' was found.<br />" );

         // search for pattern 'Now' in the beginning of
         // the string
         if ( ereg( "^Now", $search ) )
            print( "String 'Now' found at beginning
               of the line.<br />" );
```

Fig. 29.8 Using regular expressions (part 1 of 2).

```
27
28          // search for pattern 'Now' at the end of the string
29          if ( ereg( "Now$", $search ) )
30             print( "String 'Now' was found at the end
31                of the line.<br />" );
32
33          // search for any word ending in 'ow'
34          if ( ereg( "[[:<:]]([a-zA-Z]*ow)[[:>:]]", $search,
35             $match ) )
36             print( "Word found ending in 'ow': " .
37                $match[ 1 ] . "<br />" );
38
39          // search for any words beginning with 't'
40          print( "Words beginning with 't' found: ");
41
42          while ( eregi( "[[:<:]](t[[:alpha:]]+)[[:>:]]",
43             $search, $match ) ) {
44             print( $match[ 1 ] . " " );
45
46             // remove the first occurrence of a word beginning
47             // with 't' to find other instances in the string
48             $search = ereg_replace( $match[ 1 ], "", $search );
49          }
50
51          print( "<br />" );
52       ?>
53    </body>
54 </html>
```

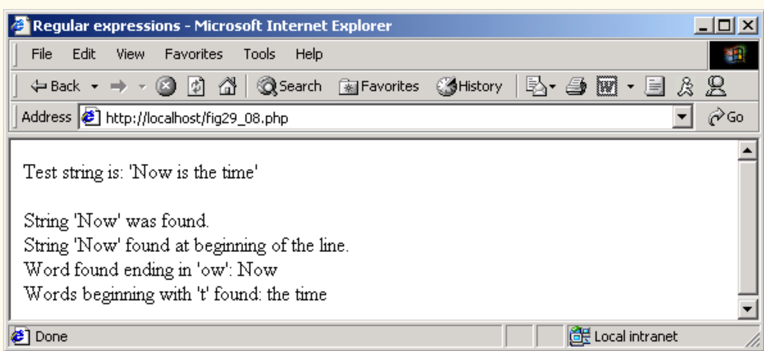


Fig. 29.8 Using regular expressions (part 2 of 2).

We begin by assigning the string **"Now is the time"** to variable **$search** (line 14). Line 19's condition calls function **ereg** to search for the *literal characters* **Now** inside variable **$search**. If the pattern is found, **ereg** returns **true**, and line 20 **print**s a message indicating that the pattern was found. We use single quotes (**''**) inside the **print** statement to emphasize the search pattern. When located inside a string, content delimited by single quotes is interpolated. If a **print** statement uses only single quotes, the content inside the single quotes is not interpolated. For example,

```
print( '$name' );
```

in a **print** statement would output **$name**. Function **ereg** takes two arguments: a regular expression pattern to search for (**Now**) and the string to search. Although case mixture and whitespace are typically significant in patterns, PHP provides *function* ***eregi*** for specifying case insensitive pattern matches.

In addition to literal characters, regular expressions can include special characters that specify patterns. For example, the *caret (****^****) special character* matches the beginning of a string. Line 24 searches the beginning of **$search** for the pattern **Now**.

The characters **$**, **^** and **.** are part of a special set of characters called *metacharacters*. A *dollar sign (****$****)* searches for the specified pattern at the end of the string (line 29). Because the pattern **Now** is not found at the end of **$search**, the body of the **if** statement (lines 30–31) is not executed. Note that **Now$** is not a variable, it is a pattern that uses **$** to search for characters **Now** at the end of a string. Another special character is the period (**.**), which matches any single character.

Lines 34–35 search (from left to right) for the first word ending with the letters **ow**. *Bracket expressions* are lists of characters enclosed in braces (**[ ]**), which match a single character from the list. Ranges can be specified by supplying the beginning and the end of the range separated by a dash (**-**). For instance, the bracket expression **[a-z]** matches any lowercase letter, and **[A-Z]** matches any uppercase letter. In this example, we combine the two to create an expression that matches any letter. The special bracket expressions ***[[:<:]]*** and ***[[:>]]*** match the beginning and end of a word, respectively.

The expression inside the parentheses, **[a-zA-Z]*ow**, matches any word ending in **ow**. It uses the *quantifier* ***** to match the preceding pattern 0 or more times. Thus, **[a-zA-Z]*ow** matches any number of characters followed by the literal characters **ow**. Figure 29.9 lists some PHP quantifiers.

Placing a pattern in parentheses stores the matched string in the array that is specified in the third argument to function **ereg**. The first parenthetical pattern matched is stored in the second array element, the second in the third array element, and so on. The first element (i.e., index 0) stores the string matched for the entire pattern. The parentheses in lines 34–35 result in **Now** being stored in variable **$match[1]**.

Quantifier	Matches
{n}	Exactly **n** times.
{m,n}	Between **m** and **n** times inclusive.
{n,}	**n** or more times.
+	One or more times (same as **{1,}**).
*****	Zero or more times (same as **{0,}**).
?	Zero or one times (same as **{0,1}**).

Fig. 29.9 Some PHP quantifiers.

Searching for multiple instances of a pattern in a string is slightly more complicated, because the **ereg** function matches only the first instance of the pattern. To find multiple instances of a given pattern, we must remove any matched instances before calling **ereg**

again. Lines 42–49 use a **while** loop and the ***ereg_replace*** *function* to find all the words in the string that begin with **t**. We will say more about this function momentarily.

The pattern used in this example, **[[:<:]](t[[:alpha:]]+)[[:>:]]**, matches any word beginning with the character **t** followed by one or more characters. The example uses the *character class* **[[:alpha:]]** to recognize any alphabetic character. This is equivalent to the **[a-zA-Z]** bracket expression that was used earlier. Figure 29.10 lists some character classes that can be matched with regular expressions.

The quantifier **+** matches one or more instances of the preceding expression. The result of the match is stored in **$match[1]**. Once a match is found, we **print** it on line 44. We then remove it from the string on line 48, using function **ereg_replace**. Function **ereg_replace** takes three arguments: the pattern to match, a string to replace the matched string and the string to search. The modified string is returned. Here, we search for the word that we matched with the regular expression, replace the word with an empty string then assign the result back to **$search**. This allows us to match any other words beginning with the character **t** in the string.

29.4 Viewing Client/Server Environment Variables

Knowledge of a client's execution environment is useful to system administrators who want to provide client-specific information. *Environment variables* contain information about a script's environment, such as the client's Web browser, the HTTP host and the HTTP connection.

Figure 29.11 generates an XHTML document that displays the values of the client's environment variables in a table. PHP stores the environment variables and their values in the ***$GLOBALS*** *array*. Iterating through this array allows us to view all the client's environment variables.

In lines 19–22, we use a **foreach** loop to **print** out the keys and values for each element in the **$GLOBALS** array. Individual array variables can be accessed directly by using an element's key from the **$GLOBALS** array as a variable. For example, to receive information about the user's browser, use the ***$HTTP_USER_AGENT*** *variable*. Figure 29.12 lists some global variables.

Character Class	Description
alnum	Alphanumeric characters (i.e., letters [a-z][A-Z] or digits [0-9]).
alpha	Word characters (i.e., letters [a-z][A-Z]).
digit	Digits.
space	Whitespace.
lower	Lowercase letters.
upper	Uppercase letters.

Fig. 29.10 Some PHP character classes.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.11: fig29_11.php                  -->
<!-- Program to display environment variables -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Environment Variables</title>
   </head>

   <body>
      <table border = "0" cellpadding = "2" cellspacing = "0"
         width = "100%">
      <?php

         // print the key and value for each element in the
         // in the $GLOBALS array
         foreach ( $GLOBALS as $key => $value )
            print( "<tr><td bgcolor = \"#11bbff\">
               <strong>$key</strong></td>
               <td>$value</td></tr>" );
      ?>
      </table>
   </body>
</html>
```

Fig. 29.11 Displaying the environment variables (part 1 of 2).

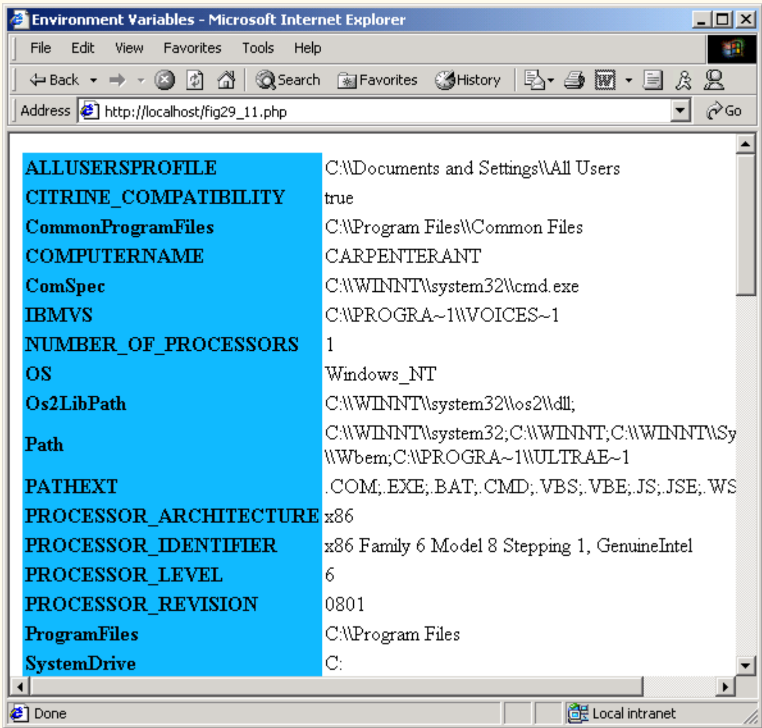


Fig. 29.11 Displaying the environment variables (part 2 of 2).

Variable Name	Description
`$HTTP_USER_AGENT`	The client's browser type.
`$REMOTE_ADDR`	The client's IP address.
`$SERVER_NAME`	Name of the server on which the script is running.
`$SERVER_ADDR`	Address of the server on which the script is running.
`$HTTP_GET_VARS`	Data posted to the server by the *get* method.
`$HTTP_POST_VARS`	Data posted to the server by the *post* method.
`$HTTP_COOKIE_VARS`	Data contained in cookies on the client's computer.
`$GLOBALS`	Array containing all global variables.

Fig. 29.12 Some environment variables.

29.5 Form Processing and Business Logic

XHTML forms enable Web pages to collect data from users and send the data to a Web server for processing. Such interaction between users and Web servers is vital to e-commerce applications, for example. Such capabilities allow users to purchase products, re-

quest information, send and receive Web-based e-mail, perform online paging and take advantage of various other online services. Figure 29.13 uses an XHTML **form** to collect information about users for the purpose of adding users to mailing lists. The type of registration form in this example could be used by a software company to acquire profile information before allowing users to download software.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.13: fig29_13.html              -->
<!-- Form for use with the form.php program -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Sample form to take user input in XHTML</title>
   </head>

   <body>

      <h1>This is a sample registration form.</h1>
      Please fill in all fields and click Register.

      <!-- post form data to form.php -->
      <form method = "post" action = "fig29_14.php">
         <img src = "images/user.gif" alt = "User" /><br />
         <span style = "color: blue">
            Please fill out the fields below.<br />
         </span>

         <!-- create four text boxes for user input -->
         <img src = "images/fname.gif" alt = "First Name" />
         <input type = "text" name = "fname" /><br />

         <img src = "images/lname.gif" alt = "Last Name" />
         <input type = "text" name = "lname" /><br />

         <img src = "images/email.gif" alt = "Email" />
         <input type = "text" name = "email" /><br />

         <img src = "images/phone.gif" alt = "Phone" />
         <input type = "text" name = "phone" /><br />

         <span style = "font-size: 10pt">
            Must be in the form (555)555-5555</span>
         <br /><br />

         <img src = "images/downloads.gif"
            alt = "Publications" /><br />

         <span style = "color: blue">
            Which book would you like information about?
         </span><br />
```

Fig. 29.13 XHTML form for gathering user input (part 1 of 3).

```

         <!-- create drop-down list containing book names -->
         <select name = "book">
            <option>Internet and WWW How to Program 2e</option>
            <option>C++ How to Program 3e</option>
            <option>Java How to Program 4e</option>
            <option>XML How to Program 1e</option>
         </select>
         <br /><br />

         <img src = "images/os.gif" alt = "Operating System" />
         <br /><span style = "color: blue">
            Which operating system are you currently using?
         <br /></span>

         <!-- create five radio buttons -->
         <input type = "radio" name = "os" value = "Windows NT"
            checked = "checked" />
            Windows NT

         <input type = "radio" name = "os" value =
            "Windows 2000" />
            Windows 2000

         <input type = "radio" name = "os" value =
            "Windows 98" />
            Windows 98<br />

         <input type = "radio" name = "os" value = "Linux" />
            Linux

         <input type = "radio" name = "os" value = "Other" />
            Other<br />

          <!-- create a submit button -->
         <input type = "submit" value = "Register" />
      </form>

   </body>
</html>
```

Fig. 29.13 XHTML form for gathering user input (part 2 of 3).

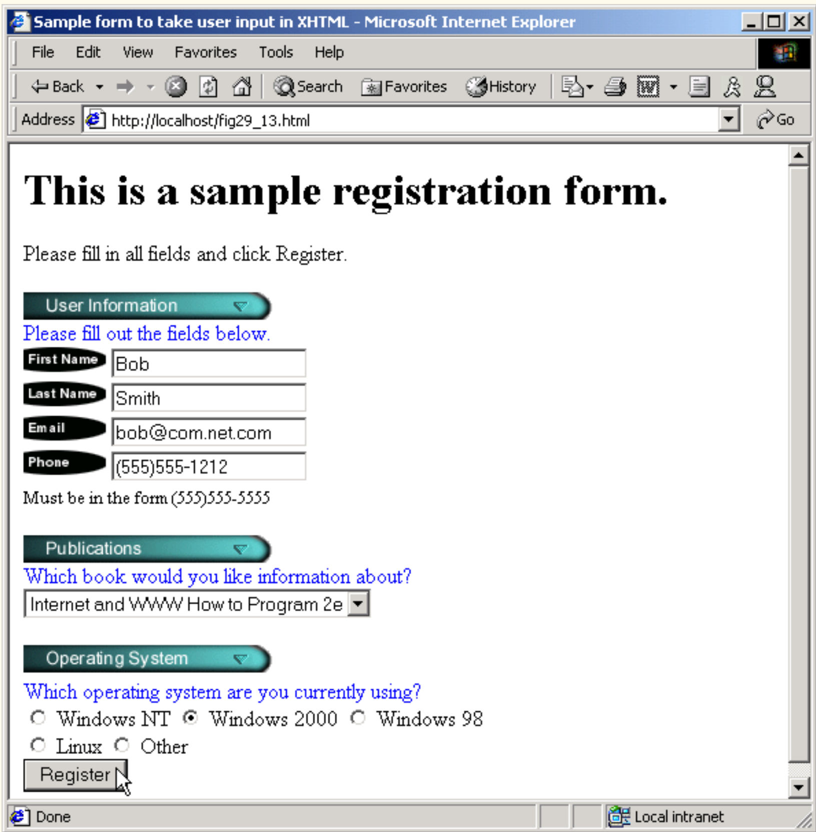


Fig. 29.13 XHTML form for gathering user input (part 3 of 3).

The **`action`** attribute of the **`form`** element (line 18) indicates that, when the user clicks **Register**, the **`form`** data will be **`post`**ed to Fig. 29.14 for processing. Using ***`method = "post"`*** appends form data to the browser request which contains the protocol (i.e., HTTP) and the requested resource's URL. Scripts located on the Web server's machine (or on a machine accessible through the network) can access the form data sent as part of the request.

We assign a unique name (e.g., **`email`**) to each of the **`form`**'s **`input`** fields. When **Register** is clicked, each field's **`name`** and **`value`** are sent to the Web server. Figure 29.14 can then accesses the submitted value for each specific field.

Good Programming Practice 29.2

*Use meaningful XHTML object names for **`input`** fields. This makes PHP scripts that retrieve **`form`** data easier to understand.*

Figure 29.14 processes the data **`post`**ed by Fig. 29.13 and sends XHTML back to the client. For each **`form`** field **`post`**ed to a PHP script, PHP creates a global variable with the same name as the field. For example, in line 32 of Fig. 29.13, an XHTML text box is cre-

ated and given the name **email**. Later in our PHP script (line 67), we access the field's value by using variable **$email**.

In lines 18–19, we determine whether the phone number entered by the user is valid. In this case, the phone number must begin with an opening parenthesis, followed by an area code, a closing parenthesis, an exchange, a hyphen and a line number. It is crucial to validate information that will be entered into databases or used in mailing lists. For example, validation can be used to ensure that credit-card numbers contain the proper number of digits before the numbers are encrypted to a merchant. The design of verifying information is called *business logic* (or *business rules*).

The expression **\(** matches the opening parenthesis of the phone number. Because we want to match the literal character **(**, we *escape* its normal meaning by preceding it with the **** character. The parentheses in the expression must be followed by three digits (**[0-9]{3}**), a closing parenthesis, three digits, a literal hyphen and four additional digits. Note that we use the **^** and **$** symbols to ensure that no extra characters appear at either end of the string.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <!-- Fig. 29.14: fig29_14.php             -->
5  <!-- Read information sent from form.html -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Form Validation</title>
10    </head>
11
12    <body style = "font-family: arial,sans-serif">
13
14       <?php
15
16          // determine if phone number is valid and print
17          // an error message if not
18          if ( !ereg( "^\([0-9]{3}\)[0-9]{3}-[0-9]{4}$",
19             $phone ) ){
20
21             print( "<p><span style = \"color: red;
22                font-size: 2em\">
23                INVALID PHONE NUMBER</span><br />
24                A valid phone number must be in the form
25                <strong>(555)555-5555</strong><br />
26                <span style = \"color: blue\">
27                Click the Back button, enter a valid phone
28                number and resubmit.<br /><br />
29                Thank You.</span></p></body></html>" );
30
31             die(); // terminate script execution
32          }
33       ?>
34
```

Fig. 29.14 Obtaining user input through forms (part 1 of 3).

```
      <p>Hi
         <span style = "color: blue">
            <strong>
               <?php print( "$fname" ); ?>
            </strong>
         </span>.
         Thank you for completing the survey.<br />

         You have been added to the
         <span style = "color: blue">
            <strong>
               <?php print( "$book " ); ?>
            </strong>
         </span>
         mailing list.
      </p>
      <strong>The following information has been saved
          in our database:</strong><br />

      <table border = "0" cellpadding = "0" cellspacing = "10">
         <tr>
            <td bgcolor = "#ffffaa">Name </td>
            <td bgcolor = "#ffffbb">Email</td>
            <td bgcolor = "#ffffcc">Phone</td>
            <td bgcolor = "#ffffdd">OS</td>
         </tr>

         <tr>
            <?php

               // print each form field's value
               print( "<td>$fname $lname</td>
                  <td>$email</td>
                  <td>$phone</td>
                  <td>$os</td>" );
            ?>
         </tr>
      </table>

      <br /><br /><br />
      <div style = "font-size: 10pt; text-align: center">
         This is only a sample form.
         You have not been added to a mailing list.
      </div>
   </body>
</html>
```

Fig. 29.14 Obtaining user input through forms (part 2 of 3).

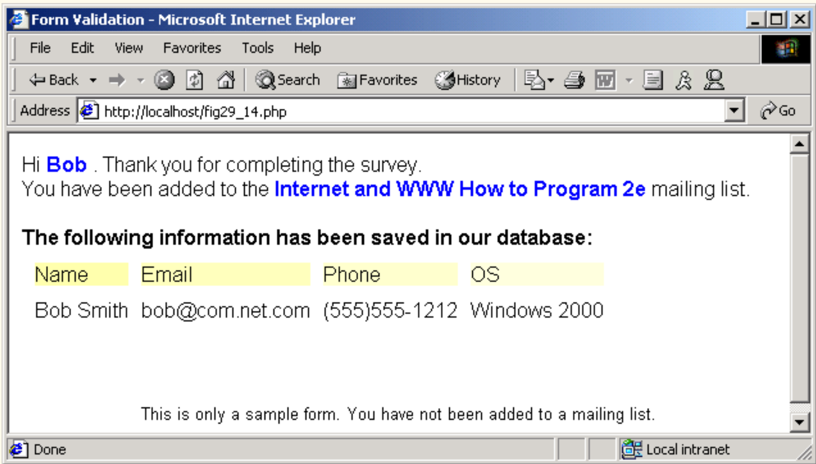


Fig. 29.14 Obtaining user input through forms (part 3 of 3).

If the regular expression is matched, then the phone number is determined to be valid, and an XHTML document is sent to the client, thanking the user for completing the form. Otherwise, the body of the **if** statement is executed, and an error message is printed.

Function ***die*** (line 31) terminates script execution. In this case, if the user did not enter a correct telephone number, we do not want to continue executing the rest of the script, so we call function **die**.

Software Engineering Observation 29.1

Use business logic to ensure that invalid information is not stored in databases. When possible, use JavaScript to validate form data while conserving server resources. However, some data, such as passwords, must be validated on the server-side.

29.6 Verifying a Username and Password

It is often desirable to have a *private Web site*—one that is accessible only to certain individuals. Implementing privacy generally involves username and password verification. Figure 29.15 presents an XHTML **form** that queries the user for a username and a password. Fields **USERNAME** and **PASSWORD** are **post**ed to the PHP script **fig29_16.php** for verification. For simplicity, we do not encrypt the data before sending it to the server. For more information regarding PHP encryption functions, visit

www.php.net/manual/en/ref.mcrypt.php

[*Note*: These functions are not available for Windows distributions of PHP.]

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.15: fig29_15.html                       -->
```

Fig. 29.15 XHTML form for obtaining a username and password (part 1 of 3).

```
<!-- XHTML form sent to fig29_16.php for verification -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Verifying a username and a password.</title>

      <style type = "text/css">
         td { background-color: #DDDDDD }
      </style>
   </head>

   <body style = "font-family: arial">
      <p style = "font-size: 13pt">
         Type in your username and password below.
         <br />
         <span style = "color: #0000FF; font-size: 10pt;
            font-weight: bold">
            Note that password will be sent as plain text
         </span>
      </p>

      <!-- post form data to fig29_16.php -->
      <form action = "fig29_16.php" method = "post">
         <br />

         <table border = "0" cellspacing = "0"
            style = "height: 90px; width: 123px;
            font-size: 10pt" cellpadding = "0">

            <tr>
               <td colspan = "3">
                  <strong>Username:</strong>
               </td>
            </tr>

            <tr>
               <td colspan = "3">
                  <input size = "40" name = "USERNAME"
                     style = "height: 22px; width: 115px" />
               </td>
            </tr>

            <tr>
               <td colspan = "3">
                  <strong>Password:</strong>
               </td>
            </tr>

            <tr>
               <td colspan = "3">
                  <input size = "40" name = "PASSWORD"
                     style = "height: 22px; width: 115px"
                     type = "password" />
               <br/></td>
```

Fig. 29.15 XHTML form for obtaining a username and password (part 2 of 3).

```
                </tr>

                <tr>
                   <td colspan = "1">
                      <input type = "submit" name = "Enter"
                         value = "Enter" style = "height: 23px;
                         width: 47px" />
                   </td>
                   <td colspan = "2">
                      <input type = "submit" name = "NewUser"
                         value = "New User"
                         style = "height: 23px" />
                   </td>
                </tr>
             </table>
          </form>
       </body>
    </html>
```

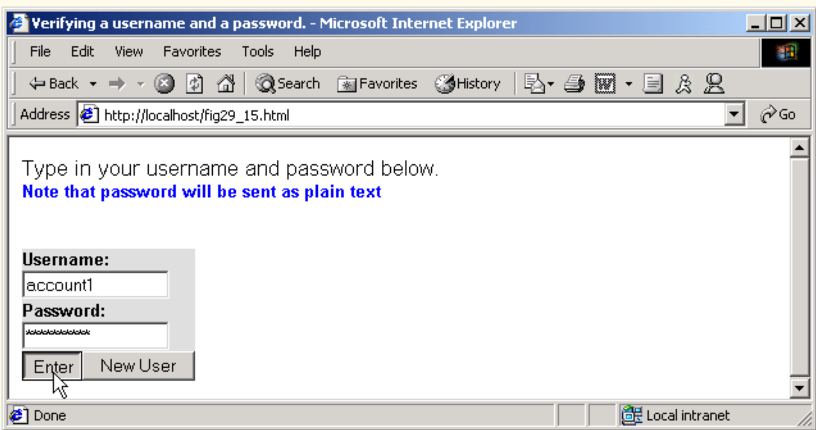


Fig. 29.15 XHTML form for obtaining a username and password (part 3 of 3).

Figure 29.16 verifies the client's username and password by querying a database. The valid user list and each user's respective password is contained within a simple text file (Fig. 29.17). Existing users are validated against this text file, and new users are appended to it.

First, lines 13–16 check whether the user has submitted a form without specifying a username or password. Variable names, when preceded by the *logical negation operator* (**!**), return **true** if they are empty or are set to **0**. *Logical operator OR* (**||**) returns **true** if either of the variables are empty or are set to **0**. If this is the case, function **fieldsBlank** is called (line 144), which notifies the client that all form fields must be completed.

We determine whether we are adding a new user (line 19 in Fig. 29.16) by calling *function* **isset** to test whether variable **$NewUser** has been set. When a user submits the XHTML form in **password.html**, the user clicks either the **New User** or **Enter** button. This sets either variable **$NewUser** or variable **$Enter**, respectively. If variable

`$NewUser` has been set, lines 22–36 are executed. If this variable has not been set, we assume the user has pressed the **Enter** button, and lines 42–75 execute.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <!-- Fig. 29.16: fig29_16.php                           -->
5  <!-- Searching a database for usernames and passwords -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <?php
10
11          // check if user has left USERNAME
12          // or PASSWORD field blank
13          if ( !$USERNAME || !$PASSWORD ) {
14             fieldsBlank();
15             die();
16          }
17
18          // check if the New User button was clicked
19          if ( isset( $NewUser ) ) {
20
21             // open fig29_17.txt for writing using append mode
22             if ( !( $file = fopen( "fig29_17.txt",
23                "append" ) ) ) {
24
25                // print error message and terminate script
26                // execution if file cannot be opened
27                print( "<title>Error</title></head><body>
28                  Could not open password file
29                  </body></html>" );
30                die();
31             }
32
33             // write username and password to file and
34             // call function userAdded
35             fputs( $file, "$USERNAME,$PASSWORD\n" );
36             userAdded( $USERNAME );
37          }
38          else {
39
40             // if a new user is not being added, open file
41             // for reading
42             if ( !( $file = fopen( "fig29_17.txt",
43                "read" ) ) ) {
44                print( "<title>Error</title></head>
45                   <body>Could not open password file
46                   </body></html>" );
47                die();
48             }
49
```

Fig. 29.16 Verifying a username and password (part 1 of 4).

```
         $userVerified = 0;

         // read each line in file and check username
         // and password
         while ( !feof( $file ) && !$userVerified ) {

            // read line from file
            $line = fgets( $file, 255 );

            // remove newline character from end of line
            $line = chop( $line );

            // split username and password
            $field = split( ",", $line, 2 );

            // verify username
            if ( $USERNAME == $field[ 0 ] ) {
               $userVerified = 1;

               // call function checkPassword to verify
               // user's password
               if ( checkPassword( $PASSWORD, $field )
                  == true )
                  accessGranted( $USERNAME );
               else
                  wrongPassword();
            }
         }

         // close text file
         fclose( $file );

         // call function accessDenied if username has
         // not been verified
         if ( !$userVerified )
            accessDenied();
      }

       // verify user password and return a boolean
      function checkPassword( $userpassword, $filedata )
      {
         if ( $userpassword == $filedata[ 1 ] )
            return true;
         else
            return false;
      }

       // print a message indicating the user has been added
      function userAdded( $name )
      {
         print( "<title>Thank You</title></head>
            <body style = \"font-family: arial;
            font-size: 1em; color: blue\">
            <strong>You have been added
            to the user list, $name.
```

Fig. 29.16 Verifying a username and password (part 2 of 4).

```
                   <br />Enjoy the site.</strong>" );
            }

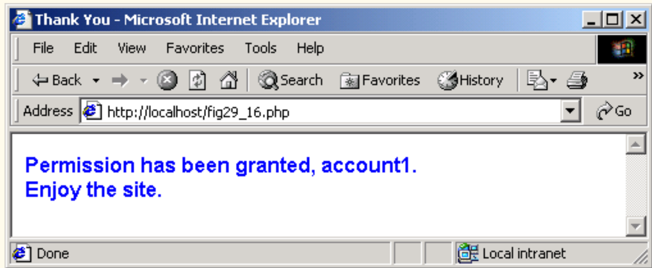
            // print a message indicating permission
            // has been granted
            function accessGranted( $name )
            {
               print( "<title>Thank You</title></head>
                  <body style = \"font-family: arial;
                  font-size: 1em; color: blue\">
                  <strong>Permission has been
                  granted, $name. <br />
                  Enjoy the site.</strong>" );
            }

            // print a message indicating password is invalid
            function wrongPassword()
            {
               print( "<title>Access Denied</title></head>
                  <body style = \"font-family: arial;
                  font-size: 1em; color: red\">
                  <strong>You entered an invalid
                  password.<br />Access has
                  been denied.</strong>" );
            }

            // print a message indicating access has been denied
            function accessDenied()
            {
               print( "<title>Access Denied</title></head>
                  <body style = \"font-family: arial;
                  font-size: 1em; color: red\">
                  <strong>
                  You were denied access to this server.
                  <br /></strong>" );
            }

            // print a message indicating that fields
            // have been left blank
            function fieldsBlank()
            {
               print( "<title>Access Denied</title></head>
                  <body style = \"font-family: arial;
                  font-size: 1em; color: red\">
                  <strong>
                  Please fill in all form fields.
                  <br /></strong>" );
            }
         ?>
      </body>
   </html>
```

Fig. 29.16 Verifying a username and password (part 3 of 4).



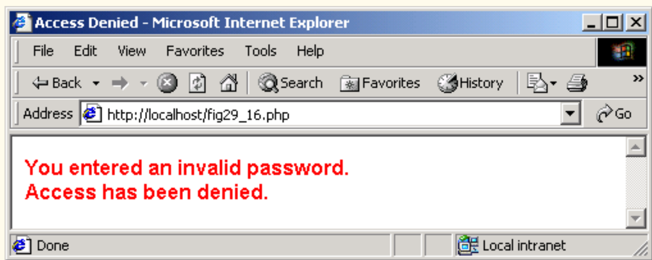


Fig. 29.16 Verifying a username and password (part 4 of 4).

```
account1,password1
account2,password2
account3,password3
account4,password4
account5,password5
account6,password6
account7,password7
account8,password8
account9,password9
account10,password10
```

Fig. 29.17 Database **fig29_17.txt** containing usernames and passwords.

To add a new user, we open the file **fig29_17.txt** by calling *function* ***fopen*** and assigning the file handle that is returned to variable **$file** (lines 22–23). A *file handle* is a number assigned to the file by the Web server for purposes of identification. Function **fopen** takes two arguments: The name of the file and the mode in which to open it. The possible modes include ***read***, ***write*** and ***append***. Here, we open the file in **append** mode, which opens it for writing, but does not write over the previous contents of the file. If an error occurs in opening the file, function **fopen** does not return a file handle and an error message is printed (lines 27–29), and script execution is terminated by calling *function* ***die*** (line 30). If the file opens properly, function **fputs** (line 35) writes the name and password to the file. To specify a new line, we use the newline character (**\n**). This places each username and password pair on a separate line in the file. On line 36, we pass the variable **$USERNAME** to function **userAdded** (line 98). Function **userAdded** **print**s a message to the client to indicate that the username and password were added to the file.

If we are not adding a new user, we open the file **fig29_17.txt** for reading. This is accomplished by using function **fopen** and assigning the file handle that is returned to variable **$file** (lines 42–43). Lines 44–47 execute if an error occurs in opening the file. The **while** loop (line 54) repeatedly executes the code enclosed in its curly braces (lines 57–75) until the test condition in parentheses evaluates to **false**. Before we enter the **while** loop, we set the value of variable **$userVerified** to **0**. In this case, the test condition (line 54) checks to ensure that the end of the file has not been reached and that the user has not been found in the password file. *Logical operator AND* (**&&**) connects the two conditions. *Function **feof***, preceded by the logical negation operator (**!**), returns **true** when there are more lines to be read in the specified file. When the logical negation operator (**!**) is applied to the **$userVerified** variable, **true** is returned if the variable is empty or is set to **0**.

Each line in **fig29_17.txt** consists of a username and password pair that is separated by a comma and followed by a newline character. A line from this file is read using function ***fgets*** (line 57) and is assigned to variable **$line**.

This function takes two arguments: The file handle to read, and the maximum number of characters to read. The function reads until a newline character is encountered, the end of the file is encountered or the number of characters read reaches one less than the number specified in the second argument.

For each line read, function ***chop*** is called (line 60) to remove the newline character from the end of the line. Then, function ***split*** is called to divide the string into substrings at the specified separator, or *delimiter* (in this case, a comma). For example, function **split** returns an array containing (**"account1"** and **"password1"**) from the first line in **fig29_17.txt**. This array is assigned to variable **$field**.

Line 66 determines whether the username entered by the user matches the one returned from the text file (stored in the variable **$field[0]**). If the condition evaluates to **true**, then the **$userVerified** variable is set to **1**, and lines 71–75 execute. On line 71, function **checkPassword** (line 89) is called to verify the user's password. Variables **$PASSWORD** and **$field** are passed to the function. Function **checkPassword** compares the user's password to the password in the file. If they match, **true** is returned (line 92), whereas **false** is returned if they do not (line 94). If the condition evaluates to **true**, then function **accessGranted** (line 110) is invoked. Variable **$USERNAME** is passed to the function, and a message notifies the client that permission has been granted. However, if the condition evaluates to **false**, then function **wrongPassword** is invoked (line 121), which notifies the client that an invalid password was entered.

When the **while** loop is complete, either as a result of matching a username or of reaching the end of the file, we are finished reading from **fig29_17.txt**. We call *function **fclose*** (line 80) to close the file. Line 84 checks whether the **$userVerified** variable is empty or has a value of **0**, which indicates that the username was not found in the **fig29_17.txt** file. If this returns **true**, function **accessDenied** is called (line 132). This function notifies the client that access to the server has been denied.

29.7 Connecting to a Database

Databases enable companies to enter the world of e-commerce by maintaining crucial data, and database connectivity allows system administrators to maintain and update such infor-

mation as user accounts, passwords, credit-card numbers, mailing lists and product inventories. PHP offers built-in support for a wide variety of databases. In this example, we use MySQL. Visit **`www.deitel.com`** to locate information on setting up a MySQL database. From a Web browser, the client enters a database field name that is sent to the Web server. The PHP script is then executed; the script builds the select query, queries the database and sends a record set in the form of XHTML to the client. The rules and syntax for writing such a query string are discussed in Chapter 17, Python Database Application Programming Interface (DB-API).

Figure 29.18 is a Web page that **`post`**s form data containing a database field to the server. The PHP script in Fig. 29.19 processes the form data.

Line 17 creates an XHTML **`form`**, specifying that the data submitted from the **`form`** will be sent to Fig. 29.19. Lines 22–28 add a select box to the **`form`**, set the name of the select box to **`select`**, and set its default selection to **`*`**. This value specifies that all records are to be retrieved from the database. Each database field is set as an option in the select box.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  
4  <!-- Fig. 29.18: fig29_18.html -->
5  <!-- Querying a MySQL Database -->
6  
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Sample Database Query</title>
10    </head>
11 
12    <body style = "background-color: #F0E68C">
13       <h2 style = "font-family: arial color: blue">
14          Querying a MySQL database.
15       </h2>
16 
17       <form method = "post" action = "database.php">
18          <p>Select a field to display:
19 
20             <!-- add a select box containing options -->
21             <!-- for SELECT query                    -->
22             <select name = "select">
23                <option selected = "selected">*</option>
24                <option>ID</option>
25                <option>Title</option>
26                <option>Category</option>
27                <option>ISBN</option>
28             </select>
29          </p>
30 
31          <input type = "submit" value = "Send Query"
32             style = "background-color: blue;
33             color: yellow; font-weight: bold" />
34       </form>
```

Fig. 29.18 Form to query a MySQL database (part 1 of 2).

```
    </body>
</html>
```

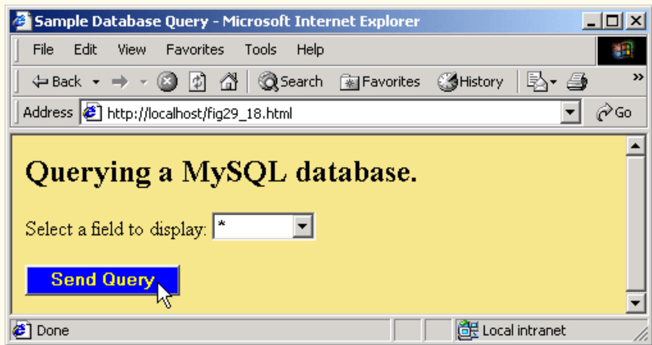


Fig. 29.18 Form to query a MySQL database (part 2 of 2).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.19: fig29_19.php        -->
<!-- Program to query database and -->
<!-- send results to client        -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Search Results</title>
   </head>

   <body style = "font-family: arial, sans-serif"
      style = "background-color: #F0E68C">
      <?php

         // build SELECT query
         $query = "SELECT " . $select . " FROM Books";

         // Connect to MySQL
         if ( !( $database = mysql_connect( "localhost",
            "httpd", "" ) ) )
            die( "Could not connect to database" );

         // open Products database
         if ( !mysql_select_db( "Products", $database ) )
            die( "Could not open Products database" );

         // query Products database
         if ( !( $result = mysql_query( $query, $database ) ) ) {
            print( "Could not execute query! <br />" );
            die( mysql_error() );
```

Fig. 29.19 Querying a database and displaying the results (part 1 of 3).

```
         }
      ?>

      <h3 style = "color: blue">
      Search Results</h3>

      <table border = "1" cellpadding = "3" cellspacing = "2"
         style = "background-color: #ADD8E6">

         <?php

            // fetch each record in result set
            for ( $counter = 0;
               $row = mysql_fetch_row( $result );
               $counter++ ){

               // build table to display results
               print( "<tr>" );

               foreach ( $row as $key => $value )
                  print( "<td>$value</td>" );

               print( "</tr>" );
            }

            mysql_close( $database );
         ?>

      </table>

      <br />Your search yielded <strong>
      <?php print( "$counter" ) ?> results.<br /><br /></strong>

      <h5>Please email comments to
         <a href = "mailto:deitel@deitel.com">
            Deitel and Associates, Inc.
         </a>
      </h5>

   </body>
</html>
```

Fig. 29.19 Querying a database and displaying the results (part 2 of 3).

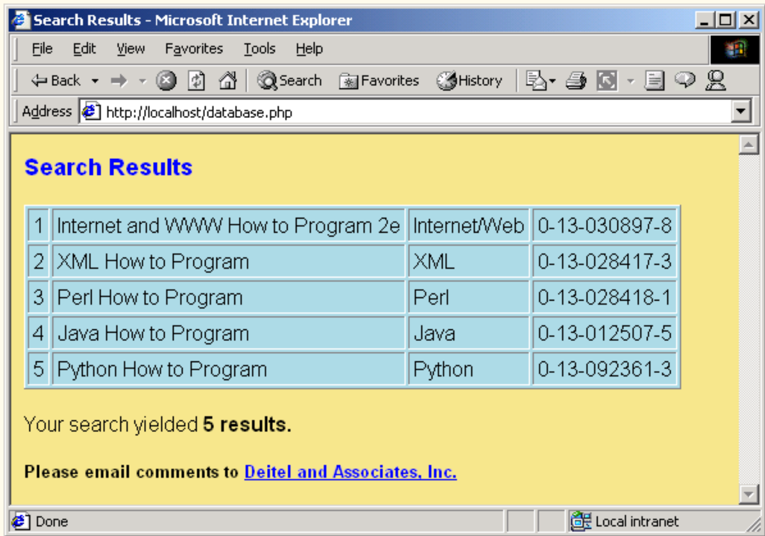


Fig. 29.19 Querying a database and displaying the results (part 3 of 3).

Figure 29.19 builds an SQL-query string with the specified field name and sending it to the database-management system. Line 18 concatenates the **post**ed field name to a **SELECT** query. Line 21 calls *function* ***mysql_connect*** to connect to the MySQL database. We pass three arguments to function **mysql_connect**: The server's hostname, a username and a password. This function returns a *database handle*—a reference to the object that is used to represent PHP's connection to the database—which we assign to variable **$database**. If the connection to MySQL fails, function **die** is called, which outputs an error message and terminates the script. Line 26 calls *function* ***mysql_select_db*** to specify the database to be queried (in this case, **Products**). Function **die** is called if the database cannot be opened. To query the database, line 30 calls *function* ***mysql_query***, specifying the query string and the database to query. Function **mysql_query** returns an object containing the result set of the query, which we assign to variable **$result**. If the query of the database fails, a message is output to the client indicating that the query failed to execute. Function **die** is then called, accepting *function* ***mysql_error*** as a parameter instead of a string message. In the event that the query fails, function **mysql_error** returns any error strings from the database. Function **mysql_query** can also be used to execute SQL statements, such as **INSERT** or **DELETE**, that do not return results.

Lines 45–56 use a **for** loop to iterate through each record in the result set while constructing an XHTML table from the results. The loop condition calls *function* ***mysql_fetch_row*** to return an array containing the elements of each row in the result set of our query (**$result**). The array is then stored in variable **$row**. Lines 52–53 use a **foreach** loop to construct individual cells for each of the elements in the row. The **foreach** loop takes the name of the array (**$row**), iterates through each index value of the array (**$key**) and stores the element in variable **$value**. Each element of the array is then printed as an individual cell. For each row retrieved, variable **$counter** is incre-

mented by one. When the end of the result set has been reached, **undef** (**false**) is returned by function **mysql_fetch_row**, which terminates the **for** loop.

After all rows of the result set have been displayed, the database is closed (line 58), and the table's closing tag is written (line 61). The number of results contained in **$counter** is printed in line 64.

29.8 Cookies

A *cookie* is a text file that a Web site stores on a client's computer to maintain information about that client during and between browsing sessions. A Web site can store a cookie on a client's computer to record user preferences and other information, which the Web site can retrieve during that client's subsequent visits. For example, many Web sites use cookies to store clients' zip codes. The Web site can retrieve the zip code from the cookie and provide weather reports and news updates tailored to the user's region. Web sites also can use cookies to track information about client activity. Analysis of information collected via cookies can reveal the popularity of various Web sites or products. In addition, marketers can use cookies to determine the effects of particular advertising campaigns.

Web sites store cookies on users' hard drives, which raises issues regarding security and privacy. Web sites should not store critical information, such as credit-card numbers or passwords in cookies, because cookies are text files that any program can read. Several cookie features address security and privacy concerns. A particular server can access only the cookies that server placed on the client. For example, a Web application running on **www.deitel.com** cannot access cookies that the Web site **www.prenhall.com/deitel** may have placed on the client's computer. A cookie also has a maximum age, after which the Web browser deletes that cookie. Users who are concerned about the privacy and security implications of cookies can disable cookies in their Web browsers. However, the disabling of cookies can prevent those users from interacting with Web sites that rely on cookies to function properly.

Microsoft Internet Explorer stores cookies as small text files on the client's hard drive. The information stored in the cookie is sent back to the Web server from which it originated whenever the user requests a Web page from that particular server. The Web server can send the client XHTML output that reflects the preferences or information that is stored in the cookie.

Figure 29.20 uses a script to write a cookie to the client's machine. The script displays an XHTML **form** that allows a user to enter a name, height and favorite color. When the user clicks the **Write Cookie** button, the script in Fig. 29.21 executes.

Software Engineering Observation 29.2

Some clients do not accept cookies. When a client declines a cookie, the browser application normally informs the client that the site may not function correctly without cookies enabled.

Software Engineering Observation 29.3

Cookies cannot be used to retrieve e-mail addresses or data from the hard drive of a client's computer.

Figure 29.21 calls *function* ***setcookie*** to set the cookies to the values passed from **cookies.html**. Function **setcookie** prints XHTML header information, therefore, it needs to be called before any other XHTML (including comments) is printed.

Function **setcookie** takes the name of the cookie to be set as the first argument, followed by the value to be stored in the cookie. For example, line 7 sets the name of the cookie to **"Name"** and the value to variable **$NAME**, which is passed to the script from Fig. 29.20. The optional third argument indicates the expiration date of the cookie. In this example, we set the cookies to expire in five days by taking the current time, which is returned by *function* ***time***, and adding the number of seconds after which the cookie should expire (**60** seconds ***** **60** minutes ***** **24** hours ***** **5** days). If no expiration date is specified, the cookie only lasts until the end of the current session, which is the total time until the user closes the browser. If only the **name** argument is passed to function **setcookie**, the cookie is deleted from the cookie database. Lines 12–37 send a Web page to the client indicating that the cookie has been written and listing the values that are stored in the cookie. Lines 34–35 provide a link to Fig. 29.24.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.20: fig29_20.html -->
<!-- Writing a Cookie          -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Writing a cookie to the client computer</title>
   </head>

   <body style = "font-family: arial, sans-serif;
      background-color: #99CCFF">

      <h2>Click Write Cookie to save your cookie data.</h2>

      <form method = "post" action = "fig29_21.php"
            style = "font-size: 10pt">
         <strong>Name:</strong><br />
         <input type = "text" name = "NAME" /><br />

         <strong>Height:</strong><br />
         <input type = "text" name = "HEIGHT" /><br />

         <strong>Favorite Color:</strong><br />
         <input type = "text" name = "COLOR" /><br />

         <input type = "submit" value = "Write Cookie"

            style = "background-color: #F0E86C; color: navy;
            font-weight: bold" /></p>
      </form>
   </body>
```

Fig. 29.20 Gathering data to be written as a cookie (part 1 of 2).

```
</html>
```

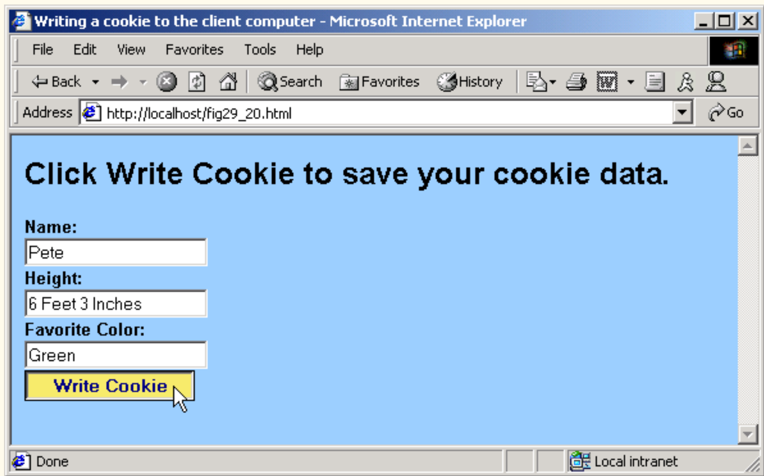


Fig. 29.20 Gathering data to be written as a cookie (part 2 of 2).

```
<?php
   // Fig. 29.21: fig29_21.php
   // Program to write a cookie to a client's machine

   // write each form field’s value to a cookie and set the
   // cookie’s expiration date
   setcookie( "Name", $NAME, time() + 60 * 60 * 24 * 5 );
   setcookie( "Height", $HEIGHT, time() + 60 * 60 * 24 * 5 );
   setcookie( "Color", $COLOR, time() + 60 * 60 * 24 * 5 );
?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Cookie Saved</title>
   </head>

   <body style = "font-family: arial, sans-serif">
      <p>The cookie has been set with the following data:</p>

      <!-- print each form field’s value -->
      <br /><span style = "color: blue">Name:</span>
         <?php print( $NAME ) ?><br />

      <span style = "color: blue">Height:</span>
         <?php print( $HEIGHT ) ?><br />
```

Fig. 29.21 Writing a cookie to the client (part 1 of 2).

```

         <span style = "color: blue">Favorite Color:</span>

         <span style = "color: <?php print( "$COLOR\">$COLOR" ) ?>
         </span><br />
         <p>Click <a href = "fig29_24.php">here</a>
            to read the saved cookie.</p>
      </body>
   </html>
```

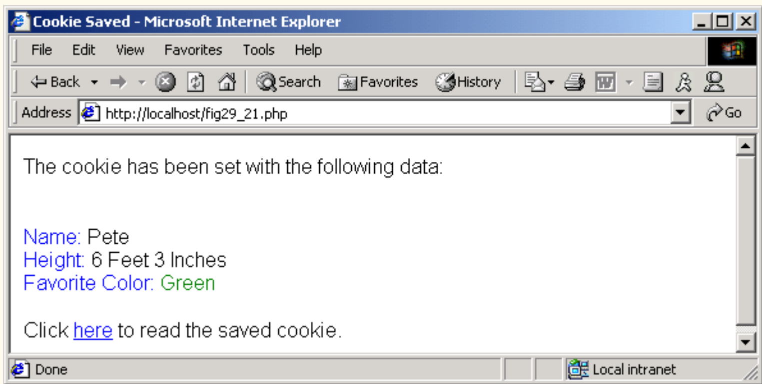


Fig. 29.21 Writing a cookie to the client (part 2 of 2).

If the client is Internet Explorer, cookies are stored in the **Cookies** directory on the client's machine. Figure 29.22 shows the contents of this directory prior to the execution of Fig. 29.21. After the cookie is written, a text file is added to the directory. In Fig. 29.23, the file **petel@localhost** appears in the **Cookies** directory.

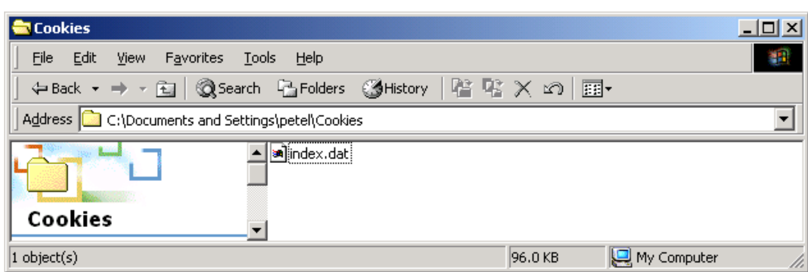


Fig. 29.22 **Cookies** directory before a cookie is written.

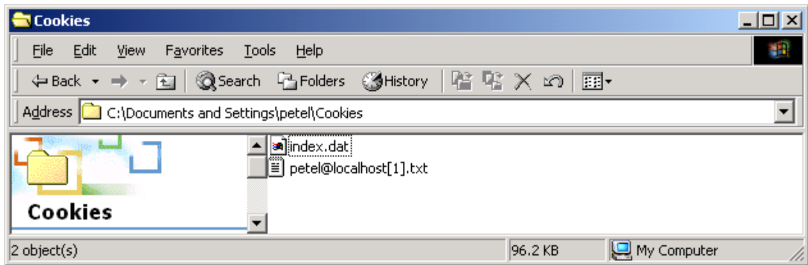


Fig. 29.23 **`Cookies`** directory after a cookie is written.

Figure 29.24 reads the cookie that is written in Fig. 29.21 and displays the cookie's information in a table.

PHP creates variables containing contents of a cookie, similar to when values are posted via forms. Thus, the next time a script is run from a location where the cookie is visible, a cookie set with the name **`"Color"`** is assigned to variable **`$Color`** along with its corresponding value. PHP also creates array **`$HTTP_COOKIE_VARS`**, which contains all the cookie values indexed by their names.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. 29.24: fig29_24.php                       -->
<!-- Program to read cookies from client's computer -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Read Cookies</title></head>

   <body style = "font-family: arial, sans-serif">

      <p>
         <strong>
            The following data is saved in a cookie on your
            computer.
         </strong>
      </p>

      <table border = "5" cellspacing = "0" cellpadding = "10">
         <?php

            // iterate through array $HTTP_COOKIE_VARS and print
            // name and value of each cookie
            foreach ( $HTTP_COOKIE_VARS as $key => $value )
               print( "<tr>
                  <td bgcolor=\"#F0E68C\">$key</td>
                  <td bgcolor=\"#FFA500\">$value</td>
                  </tr>" );
         ?>
```

Fig. 29.24 Displaying the cookie's contents (part 1 of 2).

```
30
31          </table>
32       </body>
33    </html>
```

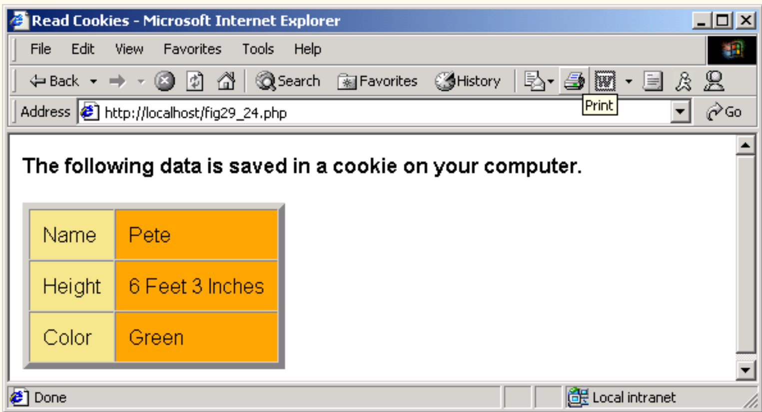


Fig. 29.24 Displaying the cookie's contents (part 2 of 2).

Lines 24–28 iterate through this array using a **`foreach`** loop, printing out the name and value of each cookie in an XHTML table. The **`foreach`** loop takes the name of the array (**`$HTTP_COOKIE_VARS`**) and iterates through each index value of the array (**`$key`**). In this case, the index value is the name of each cookie. Each element is then stored in variable **`$value`**, and these values become the individual cells of the table.

29.9 Operator Precedence

This section contains the operator precedence chart for PHP. In Fig. 29.25, the operators are shown from top to bottom in decreasing order of precedence.

Operator	Type	Associativity
`new`	constructor	none
`[]`	subscript	right to left
`~`	bitwise not	right to left
`!`	not	
`++`	increment	
`--`	decrement	
`-`	unary negative	*(This level continued at*
`@`	error control	*top of next page)*

Fig. 29.25 PHP operator precedence and associativity (part 1 of 3).

Operator	Type	Associativity
~ ! ++ -- - @	bitwise not not increment decrement unary negative error control	*(This level continued from bottom of previous page)*
* / %	multiplication division modulus	left to right
+ - .	addition subtraction concatenation	left to right
<< >>	bitwise shift left bitwise shift right	left to right
< > <= >=	less than greater than less than or equal greater than or equal	none
== != === !==	equal not equal identical not identical	none
&	bitwise AND	left to right
^	bitwise XOR	left to right
|	bitwise OR	left to right
&&	logical AND	left to right
||	logical OR	left to right
= += -= *= /= &= |= ^= .= <<= >>=	assignment addition assignment subtraction assignment multiplication assignment division assignment bitwise AND assignment bitwise OR assignment bitwise exclusive OR assignment concatenation assignment bitwise shift left assignment bitwise shift right assignment	left to right
and	logical AND	left to right

Fig. 29.25 PHP operator precedence and associativity (part 2 of 3).

Operator	Type	Associativity
xor	exclusive OR	left to right
or	logical OR	left to right
,	list	left to right

Fig. 29.25 PHP operator precedence and associativity (part 3 of 3).

29.10 Internet and World Wide Web Resources

www.php.net
This official PHP site contains the latest versions of PHP, as well as documentation, a list of FAQs, support and links to many other PHP resources.

www.zend.com
This site is the home of *Zend Technologies*, the developers of the Zend scripting engine. The site also provides code, tips and applications for PHP developers.

www.phpbuilder.com
This site contains resources for PHP developers. The site also includes a search feature and provides links to articles, code and forums.

www.phpworld.com
This site provides PHP-related resources, including articles, documentation, links and a help board.

php.resourceindex.com
This site provides access to the PHP community, helping visitors find jobs, chats, developer sites and more. The code section of the site contains scripts, functions and classes. In addition, visitors can sign up to receive e-mail updates regarding new resources.

www.phpwizard.net
This site contains resources for PHP development. It provides tutorials, links and many other resources.

phpclub.unet.ru
This Web page contains manuals, forums, links, books, databases, a FAQ list and other PHP resources.

SUMMARY

- PHP is an open-source technology that is supported by a large community of users and developers. PHP is platform independent; implementations exist for all major UNIX, Linux and Windows operating systems.
- PHP code is embedded directly into XHTML documents and provides support for a wide variety of different databases. PHP scripts typically have the file extension **.php**.
- In PHP, code is inserted in special scripting delimiters that begin with **<?php** and end with **?>**.
- Variables are preceded by the **$** special symbol. A variable is created automatically when it is first encountered by the PHP interpreter.
- PHP statements are terminated with a semicolon (**;**). Comments begin with two forward slashes (**//**). Text to the right of the slashes is ignored by the interpreter.

- When a variable is encountered inside a double-quoted (**""**) string, PHP uses interpolation to replace the variable with its associated data.
- PHP variables are multitype, meaning that they can contain different types of data— integers, floating-point numbers or strings.
- Type casting converts between data types without changing the value of the variable itself.
- The concatenation operator (**.**) appends the string on the right of the operator to the string on the left.
- Uninitialized variables have the value **undef**, which evaluates to different values, depending on the context. When **undef** is used in a numeric context, it evaluates to **0**. When **undef** is interpreted in a string context, it evaluates to an empty string (**""**).
- Strings are automatically converted to integers when they are used in arithmetic operations.
- PHP provides the capability to store data in arrays. Arrays are divided into elements that behave as individual variables.
- Individual array elements are accessed by following the array-variable name with the index number in braces (**[]**). If a value is assigned to an array that does not exist, the array is created. In addition to integer indices, arrays can also have nonnumeric indices.
- Function **count** returns the total number of elements in the array. Function **array** takes a list of arguments and returns an array. Function **array** may also be used to initialize arrays with string indices.
- Function **reset** sets the iterator to the first element of the array. Function **key** returns the index of the current element. Function **next** moves the iterator to the next element.
- The **foreach** loop is a control structure that is specifically designed for iterating through arrays.
- Text manipulation in PHP is usually done with regular expressions—a series of characters that serve as pattern-matching templates (or search criteria) in strings, text files and databases. This feature allows complex searching and string processing to be performed using relatively simple expressions.
- Function **strcmp** compares two strings. If the first string alphabetically precedes the second string, **-1** is returned. If the strings are equal, **0** is returned. If the first string alphabetically follows the second string, **1** is returned.
- Relational operators (**==**, **!=**, **<**, **<=**, **>** and **>=**) can be used to compare strings. These operators can also be used for numerical comparison of integers and doubles.
- For more powerful string comparisons, PHP provides functions **ereg** and **preg_match**, which use regular expressions to search a string for a specified pattern.
- Function **ereg** uses POSIX extended regular expressions, whereas function **preg_match** provides Perl compatible regular expressions.
- The caret (**^**) matches the beginning of a string. A dollar sign (**$**) searches for the specified pattern at the end of the string. The period (**.**) is a special character that is used to match any single character. The **** character is an escape character in regular expressions.
- Bracket expressions are lists of characters enclosed in square brackets (**[ ]**) that match a single character from the list. Ranges can be specified by supplying the beginning and the end of the range separated by a dash (**-**).
- The special bracket expressions **[[:<:]]** and **[[:>:]]** match the beginning and end of a word.
- Character class **[[:alpha:]]** matches any alphabetic character.
- The quantifier **+** matches one or more instances of the preceding expression.

- Function **ereg_replace** takes three arguments: The pattern to search, a string to replace the matched string and the string to search.
- PHP stores environment variables and their values in the **$GLOBALS** array. Individual array variables can be accessed directly by using an element's key from the **$GLOBALS** array as a variable.
- For each **form** field posted to a PHP script, PHP creates a variable with the same name as the field.
- Function **die** terminates script execution.
- Passing a string argument to the **die** function prints that string a message before stopping program execution.
- Function **isset** tests whether a variable has been set.
- Function **fopen** opens a text file.
- A file handle is a number that the server assigns to the file and is used when the server accesses the file.
- Function **fopen** takes two arguments: The name of the file and the mode in which to open the file. The possible modes include **read**, **write** and **append**.
- Function **feof**, preceded by the logical negation operator (**!**), returns **true** when there are more lines to be read in a specified file.
- A line from a text file is read using function **fgets**. This function takes two arguments: The file handle to read and the maximum number of characters to read.
- Function **chop** removes newline characters from the end of a line. Function **split** divides a string into substrings at the specified separator or delimiter. Function **fclose** closes a file.
- Function **mysql_connect** connects to a MySQL database. This function returns a database handle—a reference to the object which is used to represent PHP's connection to the database. Function **mysql_query** returns an object that contains the result set of the query. Function **mysql_error** returns any error strings from the database if the query fails. Function **mysql_fetch_row** returns an array that contains the elements of each row in the result set of a query.
- Cookies maintain state information for a particular client who uses a Web browser. Cookies are often used to record user preferences or other information that will be retrieved during a client's subsequent visits to a Web site. On the server side, cookies can be used to track information about client activity.
- The data stored in the cookie is sent back to the Web server from which it originated whenever the user requests a Web page from that particular server.
- Function **setcookie** sets a cookie. Function **setcookie** takes as the first argument the name of the cookie to be set, followed by the value to be stored in the cookie.
- PHP creates variables containing contents of a cookie, similar to when values are posted via forms.
- PHP creates array **$HTTP_COOKIE_VARS**, which contains all the cookie values indexed by their names.

TERMINOLOGY

$ metacharacter
$GLOBALS variable
$HTTP_COOKIE_VARS
append
array function
array_splice function
as
assignment operator
backslash
bracket expression
caret metacharacter (**^**) in PHP
character class
chomp function
comparison operator

concatenation operator
count function
current function
database connectivity
database handle
delimiter
die function
doubleval function
environment variable
equality operator
ereg function
ereg_replace function
eregi function
fclose function
feof function
fgets function
filehandle
fopen function
foreach loop
fputs function
HTTP connection
HTTP host
Hypertext Preprocessor
index value
interpolation
intval function
isset function
key function
literal character
logical AND operator
logical negation operator (**!**)
metacharacter
MySQL
mysql_connect function
mysql_error function
mysql_fetch_row function
mysql_query function
mysql_selectdb function
newline character
next function
parenthetical memory in PHP
Perl compatible regular expression
PHP (Hypertext Preprocessor)
PHP comment
PHP keyword
pos function
POSIX extended regular expression
preg_match function
print function
printf function
quantifier
read
regular expression
reset
result set
setcookie function
settype function
split function
SQL query string
strcmp function
string context
strval function
typecasting operator
undef
validation
Web server
while loop
write

SELF-REVIEW EXERCISES

29.1 State whether the following are *true* or *false*. If *false*, explain why.

a) PHP code is embedded directly into XHTML.
b) PHP function names are case sensitive.
c) The **strval** function permanently changes the type of a variable into a string.
d) Conversion between data types happens automatically when a variable is used in a context that requires a different data type.
e) The **foreach** loop is a control structure that is designed specifically for iterating over arrays.
f) Relational operators can be used for alphabetic and numeric comparison.
g) The quantifier **+**, when used in a regular expression, matches any number of the preceding pattern.
h) Opening a file in **append** mode causes the file to be overwritten.
i) Cookies are stored on the server computer.
j) The ***** arithmetic operator has higher precedence than the **+** operator.

29.2 Fill in the blanks in each of the following statements:

a) PHP scripts typically have the file extension __________.
b) The two numeric data types that PHP variables can store are __________ and __________.
c) In PHP, uninitialized variables have the value __________.
d) __________ are divided into individual elements, each of which act like individual variables.
e) Function __________ returns the total number of elements in an array.
f) To use Perl compatible regular expressions, use the __________ function.
g) A __________ in a regular expression matches a predefined set of characters.
h) PHP stores all global variables in array __________.
i) Function __________ terminates script execution.
j) __________ maintain state information on a client's computer.

ANSWERS TO SELF-REVIEW EXERCISES

29.1 a) True. b) False. Function names are not case sensitive. c) False. The **strval** function returns the converted value, but does not affect the original variable. d) True. e) True. f) True. g) False. The quantifier **+** matches one or more of the preceding patterns. h) False. Opening a file in **write** mode causes the file to be overwritten. i) False. Cookies are stored on the client's computer. j) True.

29.1 a) **.php**. b) integers, double. c) **undef**. d) Arrays.
e) **count**. f) **preg_match**. g) character class. h) **$GLOBALS**. i) **die**. j) Cookies.

EXERCISES

29.3 Write a PHP program named **states.php** that creates a scalar value **$states** with the value **"Mississippi Alabama Texas Massachusetts Kansas"**. Write a program that does the following:

a) Search for a word in scalar **$states** that ends in **xas**. Store this word in element 0 of an array named **$statesArray**.
b) Search for a word in **$states** that begins with **k** and ends in **s**. Perform a case-insensitive comparison. Store this word in element 1 of **$statesArray**.
c) Search for a word in **$states** that begins with **M** and ends in **s**. Store this element in element 2 of the array.
d) Search for a word in **$states** that ends in **a**. Store this word in element 3 of the array.
e) Search for a word in **$states** at the beginning of the string that starts with **M**. Store this word in element 4 of the array.
f) Output the array **$statesArray** to the screen.

29.4 In the text, we presented environment variables. Develop a program that determines whether the client is using Internet Explorer. If so, determine the version number and send that information back to the client.

29.5 Modify the program in Fig. 29.14 to save information sent to the server into a text file. Each time a user submits a form, open the text file and print the file's contents.

29.6 Write a PHP program that tests whether an e-mail address is input correctly. Verify that the input begins with series of characters, followed by the **@** character, another series of characters, a period (**.**) and a final series of characters. Test your program, using both valid and invalid email addresses.

29.7 Using environment variables, write a program that logs the address (obtained with the **REMOTE_ADDR** environment variable) requesting information from the Web server.

29.8 Write a PHP program that obtains a URL and a description of that URL from a user and stores the information into a database using **MySQL**. The database should be named **URLs**, and the table should be named **Urltable**. The first field of the database, which is named **URL**, should contain an actual URL, and the second, which is named **Description**, should contain a description of that URL. Use **www.deitel.com** as the first URL, and input **Cool site!** as its description. The second URL should be **www.php.net**, and the description should be **The official PHP site**. After each new URL is submitted, print the complete results of the database in a table.

WORKS CITED

1. S.S. Bakken, et al., "Introduction to PHP," 17 April 2000 **<www.zend.com/zend/hof/rasmus.php>**.

2. S.S. Bakken, et al., "A Brief History of PHP," January 2001 **<www.php.net/manual/en/intro-history.php>**.s

(***SOLUTIONS***)

SELF-REVIEW EXERCISES

29.1 State whether the following are *true* or *false*. If *false*, explain why.

a) PHP code is embedded directly into XHTML.
ANS: True.

b) PHP function names are case sensitive.
ANS: False. Function names are not case sensitive.

c) The **strval** function permanently changes the type of a variable into a string.
ANS: False. The **strval** function returns the converted value, but does not affect the original variable.

d) Conversion between data types happens automatically when a variable is used in a context that requires a different data type.
ANS: True.

e) The **foreach** loop is a control structure that is designed specifically for iterating over arrays.
ANS: True.

f) Relational operators can be used for alphabetic and numeric comparison.
ANS: True.

g) The quantifier **+**, when used in a regular expression, matches any number of the preceding pattern.
ANS: False. The quantifier **+** matches one or more of the preceding patterns.

h) Opening a file in **append** mode causes the file to be overwritten.
ANS: False. Opening a file in **write** mode causes the file to be overwritten.

i) Cookies are stored on the server computer.
ANS: False. Cookies are stored on the client's computer.

j) The ***** arithmetic operator has higher precedence than the **+** operator.
ANS: True.

29.2 Fill in the blanks in each of the following statements:

a) PHP scripts typically have the file extension __________.
ANS: .php.

b) The two numeric data types that PHP variables can store are __________ and __________.

ANS: integers, doubles.

c) In PHP, uninitialized variables have the value __________.

ANS: **undef**.

d) __________ are divided into individual elements, each of which act like individual variables.

ANS: Arrays.

e) Function __________ returns the total number of elements in an array.

ANS: **count**.

f) To use Perl compatible regular expressions, use the __________ function.

ANS: **preg_match**.

g) A __________ in a regular expression matches a predefined set of characters.

ANS: character class.

h) PHP stores all global variables in array __________.

ANS: **$GLOBALS**.

i) Function __________ terminates script execution.

ANS: **die**.

j) __________ maintain state information on a client's computer.

ANS: Cookies.

EXERCISES

29.3 Write a PHP program named **states.php** that creates a scalar value **$states** with the value **"Mississippi Alabama Texas Massachusetts Kansas"**. Write a program that does the following:

a) Search for a word in scalar **$states** that ends in **xas**. Store this word in element 0 of an array named **$statesArray**.
b) Search for a word in **$states** that begins with **k** and ends in **s**. Perform a case-insensitive comparison. Store this word in element 1 of **$statesArray**.
c) Search for a word in **$states** that begins with **M** and ends in **s**. Store this element in element 2 of the array.
d) Search for a word in **$states** that ends in **a**. Store this word in element 3 of the array.
e) Search for a word in **$states** at the beginning of the string that starts with **M**. Store this word in element 4 of the array.
f) Output the array **$statesArray** to the screen.

ANS:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.3: ex29_03.php -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Exercise 29.3</title></head>

   <body>
      <?php
         $states =
            "Mississippi Alabama Texas Massachusetts Kansas";

         if ( eregi( "[[:<:]]([[:alpha:]]+xas)[[:>:]]", $states,
            $matches ) )
```

```
                  $statesArray[ 0 ] = $matches[ 1 ];

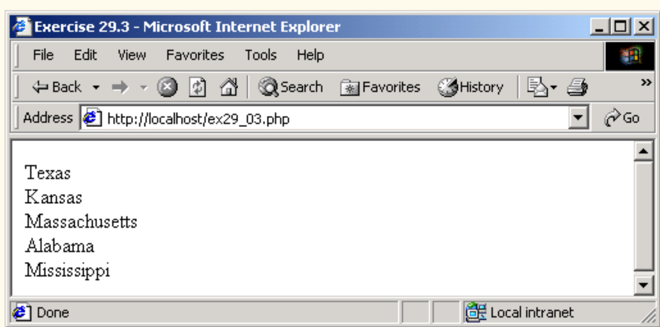
               if ( eregi( "[[:<:]](k[[:alpha:]]+s)[[:>:]]", $states,
                  $matches ) )
                  $statesArray[ 1 ] = $matches[ 1 ];

               if ( eregi( "[[:<:]](M[[:alpha:]]+s)[[:>:]]", $states,
                  $matches ) )
                  $statesArray[ 2 ] = $matches[ 1 ];

               if ( eregi( "[[:<:]]([[:alpha:]]+a)[[:>:]]", $states,
                  $matches ) )
                  $statesArray[ 3 ] = $matches[ 1 ];

               if ( eregi( "^(M[[:alpha:]]+)[[:>:]]", $states,
                  $matches ) )
                  $statesArray[ 4 ] = $matches[ 1 ];

               foreach ( $statesArray as $key => $value )
                  print( "$value <br />" );
            ?>
      </body>
   </html>
```



29.4 In the text, we presented environment variables. Develop a program that determines whether the client is using Internet Explorer. If so, determine the version number and send that information back to the client.

ANS:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.4: ex29_04.php                -->
<!-- Program to determine version number of IE -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Exercise 29.4</title></head>

   <body>
      <?php
```

```
         if ( eregi( "MSIE ([[:digit:]\.?[[:digit:]]*)",
            $HTTP_USER_AGENT, $matches ) )
            print(
               "You are using Microsoft Internet Explorer
               version " . $matches[ 1 ] );
         else
            print( "You are not running Microsoft Internet
               Explorer." );
      ?>
   </body>
</html>
```



29.5 Modify the program in Fig. 29.14 to save information sent to the server into a text file. Each time a user submits a form, open the text file and print the file's contents.

ANS:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.5: ex29_05.php              -->
<!-- Program to save information to comma    -->
<!-- delimited text file                     -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Exercise 29.5</title>
   </head>

   <body style = "font-family: arial,sans-serif">

      <?php

         // determine if the phone number is valid and print
         // an error message if it is not
         if ( !ereg( "^\([0-9]{3}\)[0-9]{3}-[0-9]{4}$",
            $phone ) ){

            print( "<p><span style = \"color: red;
               font-size: 2em\">
               INVALID PHONE NUMBER</span><br />
               A valid phone number must be in the form
               <strong>(555)555-5555</strong><br />
```

```
                  <span style = \"color: blue\">
                  Click the Back button, enter a valid phone
                  number and resubmit.<br /><br />
                  Thank You.</span></p></body></html>" );

               die(); // terminate script execution
            }

            // Write data to a text file
            if ( $file = fopen( "records.txt", "append" ) ) {
               fputs(
                  $file, "$fname, $lname, $email, $phone, $os\n" );
               fclose( $file );
            }
            else
               print( "Error opening records.txt,
                       information not recorded." );
         ?>

         <p>Hi
            <span style = "color: blue">
               <strong>
                  <?php print( "$fname" ); ?>
               </strong>
            </span>.
            Thank you for completing the survey.<br />

            You have been added to the
            <span style = "color: blue">
               <strong>
                  <?php print( "$book " ); ?>
               </strong>
            </span>
            mailing list.
         </p>
         <strong>The following information has been saved
             in our database:</strong><br />

         <table border = "0" cellpadding = "0" cellspacing = "10">
            <tr>
                <td bgcolor = "#ffffaa">Name </td>
                <td bgcolor = "#ffffbb">Email</td>
                <td bgcolor = "#ffffcc">Phone</td>
                <td bgcolor = "#ffffdd">OS</td>
            </tr>

               <?php

               // open file
               if ( !( $file = fopen( "records.txt",
                  "read" ) ) ) {
                  print( "<title>Error</title></head>
                     <body>Could not open password file
                     </body></html>" );
                  die();
```

```
            }

            // read each line in file
            while ( !feof( $file ) ) {

               // read line from file
               $line = fgets( $file, 255 );

               // remove newline character from end of line
               $line = chop( $line );

               // split each value
               $field = split( ",", $line, 5 );

               // print text file data
               print( "<tr><td>$field[0] $field[1]</td>
                  <td>$field[2]</td>
                  <td>$field[3]</td>
                  <td>$field[4]</td></tr>" );

            }

            // close text file
            fclose( $file );

            ?>
      </table>

      <br /><br /><br />
      <div style = "font-size: 10pt; text-align: center">
         This is only a sample form.
         You have not been added to a mailing list.
      </div>
   </body>
</html>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.5: ex29_05.html           -->
<!-- Form for use with ex29_05.php program -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Sample form to take user input in XHTML</title>
   </head>

   <body>

      <h1>This is a sample registration form.</h1>
      <p>Please fill in all fields and click Register.</p>

      <!-- post form data to ex29_05.php -->
      <form method = "post" action = "ex29_05.php">
```

```
            <img src = "images/user.gif" alt = "User" /><br />

            <span style = "color: blue">
               Please fill out the fields below.<br />
            </span>

            <img src = "images/fname.gif" alt = "First Name" />
            <input type = "text" name = "fname" /><br />

            <img src = "images/lname.gif" alt = "Last Name" />
            <input type = "text" name = "lname" /><br />

            <img src = "images/email.gif" alt = "Email" />
            <input type = "text" name = "email" /><br />

            <img src = "images/phone.gif" alt = "Phone" />
            <input type = "text" name = "phone" /><br />

            <span style = "font-size: 10pt">
               Must be in the form (555)555-5555</span>
            </span>
            <br /><br />

            <img src = "images/downloads.gif"
               alt = "Publications" /><br />

            <span style = "color: blue">
               Which book would you like information about?
            </span><br />

            <select name = "book">
               <option>Internet and WWW How to Program 2e</option>
               <option>C++ How to Program 3e</option>
               <option>Java How to Program 4e</option>
               <option>XML How to Program 1e</option>
            </select>
            <br /><br />

            <img src = "images/os.gif" alt = "Operating System" />
            <br /><span style = "color: blue">
               Which operating system are you currently using?
            <br /></span>

            <input type = "radio" name = "os" value = "Windows NT"
               checked = "checked" />
               Windows NT

            <input type = "radio" name = "os" value =
               "Windows 2000" />
               Windows 2000

            <input type = "radio" name = "os" value =
               "Windows 98" />
               Windows 98<br />

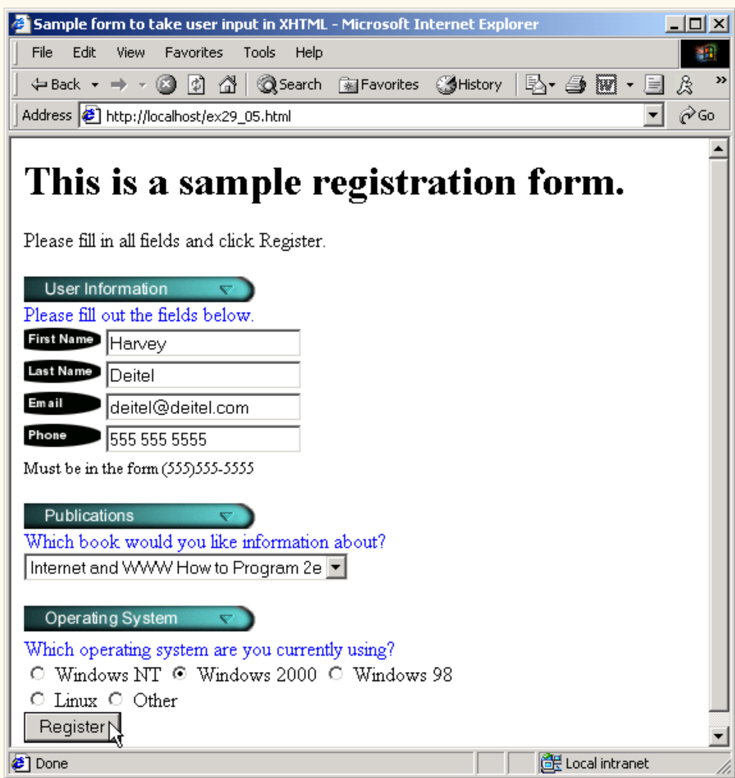
```

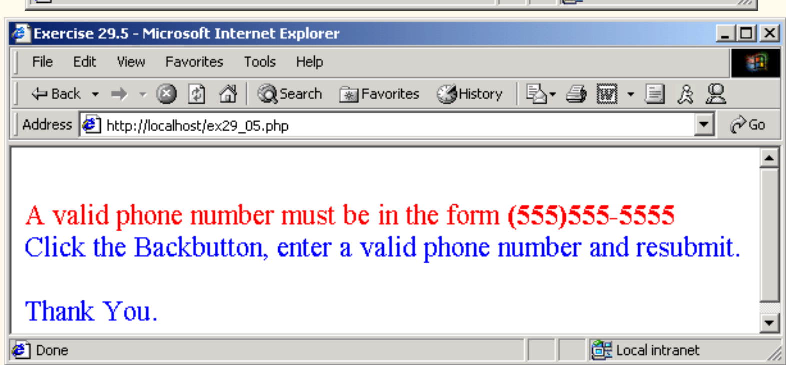
```
               <input type = "radio" name = "os" value = "Linux" />
                  Linux

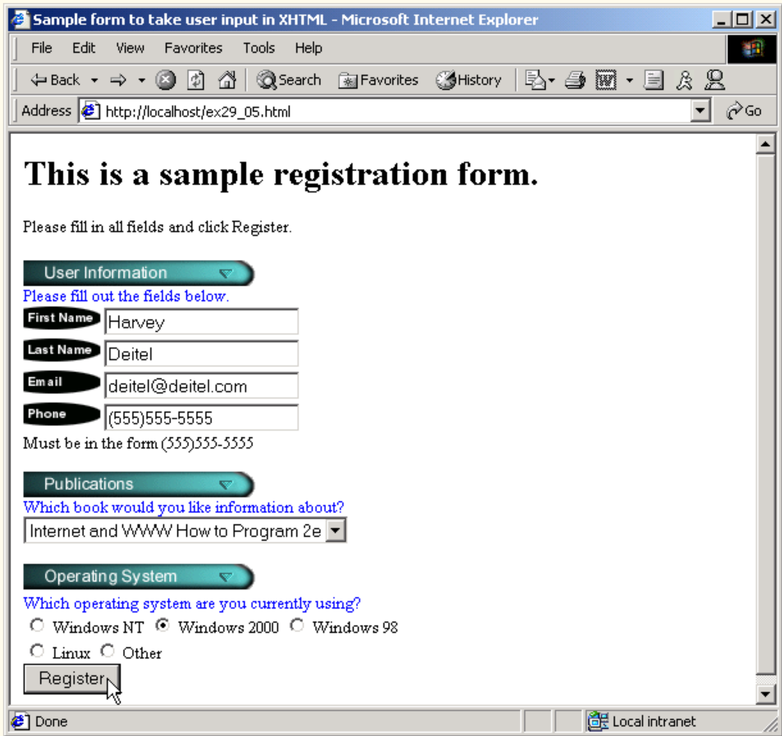
               <input type = "radio" name = "os" value = "Other" />
                  Other<br />

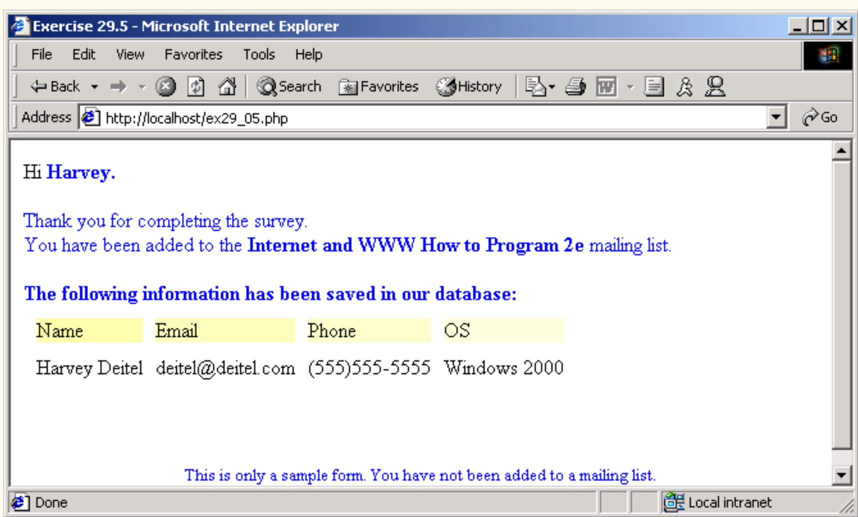
               <input type = "submit" value = "Register" />
            </form>

         </body>
      </html>
```









29.6 Write a PHP program that tests whether an e-mail address is input correctly. Verify that the input begins with series of characters, followed by the `@` character, another series of characters, a period (`.`) and a final series of characters. Test your program, using both valid and invalid email addresses.

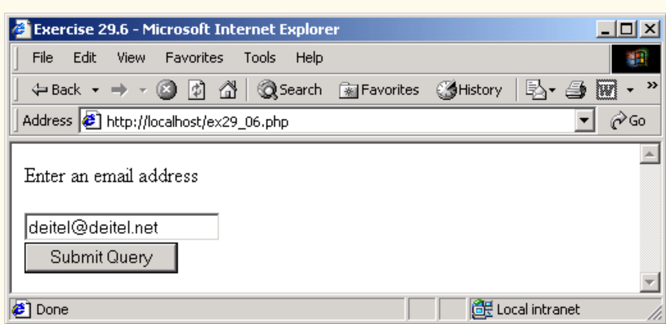
ANS:

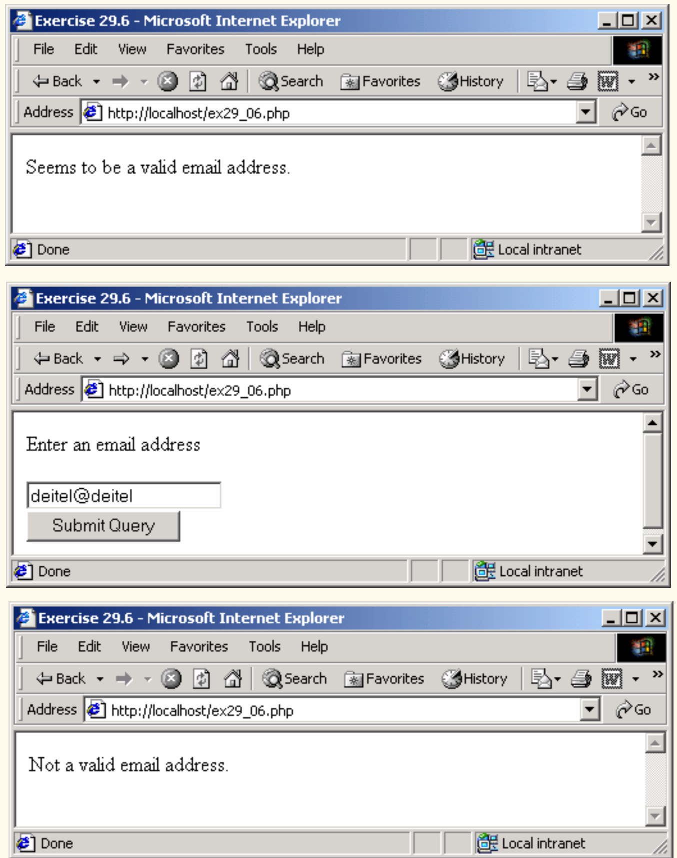
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.6: ex29_06.php -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Exercise 29.6</title></head>
   <body>
      <?php
         if ( isset( $email ) ) {

            if (
               ereg(
                  "^[[:alpha:]]+@[[:alpha:]]+\.[[:alpha:]]+$",
                     $email ) )
               print(
                  "Seems to be a valid email address.<br />" );
            else
               print( "Not a valid email address.<br />" );
         }
         else {
            print(
               "Enter an email address <form method = \"post\"
               action = \"ex29_11.php\"><input type = \"text\"
               name = \"email\" /><br />
               <input type = \"submit\" name = \"Submit\" />
               </form>" );
         }
      ?>
   </body>
</html>
```





29.7 Using environment variables, write a program that logs the address (obtained with the **REMOTE_ADDR** environment variable) requesting information from the Web server.

ANS:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.7: ex29_07.php -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Exercise 29.7</title></head>

   <body>

      <?php
         if ( $file = fopen( "ex29_07.txt", "a" ) ) {
            fputs( $file, "$REMOTE_ADDR\n" );
            fclose( $file );
            print(
```

```
                     "Your address, $REMOTE_ADDR, has been recorded." );
               }
               else
                  print( "Could not open ex29_07.txt" );
            ?>

      </body>
   </html>
```



29.8 Write a PHP program that obtains a URL and a description of that URL from a user and stores the information into a database using **MySQL**. The database should be named **URLs**, and the table should be named **Urltable**. The first field of the database, which is named **URL**, should contain an actual URL, and the second, which is named **Description**, should contain a description of that URL. Use **www.deitel.com** as the first URL, and input **Cool site!** as its description. The second URL should be **www.php.net**, and the description should be **The official PHP site**. After each new URL is submitted, print the complete results of the database in a table.

ANS:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.8: ex29_08a.php -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Exercise 29.8</title>

      <style type = "text/css">

         .blue    { color: blue; font-weight: bold }
         p        { font-family: arial, sans-serif }

      </style>
   </head>

   <body>
      <strong>
         Please enter a URL and a description.
      </strong>

      <form method = "post" action = "ex29_08b.php">
```

```
         <span class = "blue">
            URL
         </span><br />

         <input type = "text" name = "site" size = "40"
            value = "http://" /><br /><br />

         <span class = "blue">
            Description of the URL
         </span><br />

         <input type = "text" name = "description"
            size = "50" /><br /><br />
         <input type = "submit" value = "Submit" />
      </form>
   </body>
</html>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 29.8: ex29_08b.php -->

   <?php
      if ( !( $database = mysql_connect(
         "localhost", "httpd", "" ) ) )
         die( "Could not connect to database" );

      if ( !mysql_select_db( "URLs", $database ) )
         die( "Could not open URL database" );

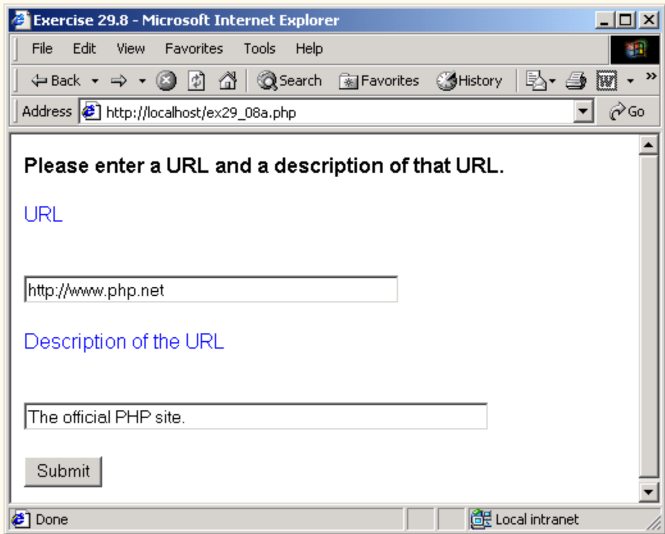
      $query = "INSERT INTO Urltable ( URL, Description )";
      $query .= "VALUES ( '$site', '$description' )";
      if ( !( $result = mysql_query( $query, $database ) ) )
      {
         print( "Could not execute query! <br />" );
         die( mysql_error() );
      }
   ?>

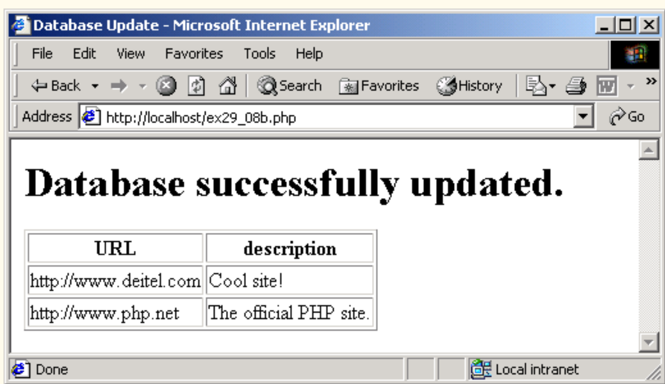
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Database Update</title></head>

   <body>

      <h1>Database successfully updated.</h1>

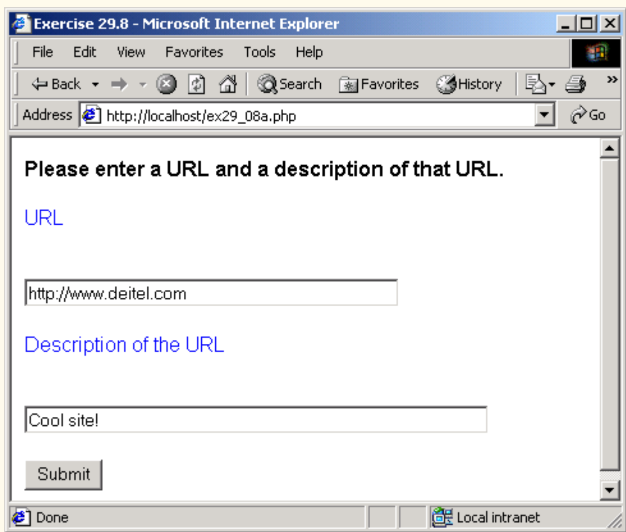
      <table border = "1">
         <tr>
            <th>URL</th><th>Description</th>
         </tr>
         <?php
            if ( !( $result = mysql_query( "SELECT URL,
               Description FROM Urltable", $database ) ) )
```

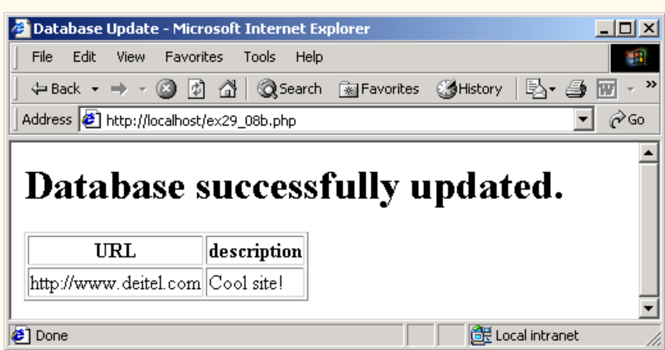

Exercise 29.8 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ex29_08a.php
Go
Please enter a URL and a description of that URL.
URL
http://www.php.net
Description of the URL
The official PHP site.
Submit
Done
Local intranet


Database Update - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ex29_08b.php
Go
Database successfully updated.
URL
description
http://www.deitel.com
Cool site!
http://www.php.net
The official PHP site.
Done
Local intranet

```
                  {
                     print( "Could not execute query! <br />" );
                     die( mysql_error() );
                  }

                  while ( $row = mysql_fetch_row( $result ) )
                     print( "<tr><td>" . $row[ 0 ] . "</td><td>"
                        . $row[ 1 ] . "</td></tr>" );
               ?>
            </table>
         </body>
      </html>
```





Symbols

A

B

C

D

E

F

G

H

I